VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Komponenta výukového serveru MAV - Ohodnocené přechodové systémy

Component of Teaching Server for Modeling and Verification - Labelled Transition Systems

2020 Richard Vašek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Richard Vašek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Komponenta výukového serveru MAV - Ohodnocené přechodové systémy
Component of Teaching Server for Modeling and Verification - Labelled Transition Systems

Jazyk vypracování: čeština

Zásady pro vypracování:

V rámci diplomových a bakalařských prací vzniká výukový server pro předměty teoretické informatiky. Jedná se o sadu dynamických webových stránek umožňujících studentům pochopení různých typů úloh a problémů tím, že si mohou zadat na stránce libovolné zadání a zobrazí se jim řešení včetně postupu. Cílem této práce je vytvořit komponentu pro výuku vybraných problémů z oblasti formální verifikace systémů.

1. Nastudujte si konečné ohodnocené přechodové systémy (LTS), bisimulační ekvivalenci a Hennessy-Milner logiku (HML).
2. Vytvořte dynamické webové stránky umožňující uživateli následující:
a) Zadat konečný LTS grafem.
b) Zobrazit si pro vybrané dva stavy zda jsou ekvivalentní podle bisimulační ekvivalence. V kladném případě zobrazit tuto ekvivalenci.
c) Zobrazit si popis vítězné strategie vítězného hráče v bisimulační hře.
d) Zadat formuli HML s rekurzí a zobrazit si podle denotační sémantiky množiny stavů, ve kterých platí jednotlivé podformule a celá formule.
3. Vytvořte i ukázkové vstupy tak, aby si vše uživatel mohl vyzkoušet i bez zadávání vlastních vstupů.

Seznam doporučené odborné literatury:

[1] Luca Aceto, Anna Ingólfsdóttir, Kim G. Larsen and Jiří Srba: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, August 2007.
[2] Christel Baier, Joost-Pieter Katoen: Principles of Model Checking, The MIT Press, 2008

Další literatura dle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Martin Kot, Ph.D.**

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2020 .....

Abstrakt

Ověřování bisimulační ekvivalence ohodnocených přechodových systémů patří mezi standardní techniky verifikace systémů. Cílem je vytvořit verifikační nástroj, který tuto bisimulační ekvivalenci ověřuje na vstupech zadaných pomocí grafického editoru a dále dovede na stejných vstupech poskytnout interaktivní hraní bisimulačních her a vyhodnocování formulí Hennessy-Milner logiky. V tomto dokumentu je popsána tvorba tohoto nástroje, včetně popisu a definic pojmů potřebných k pochopení použitých algoritmů. Korektnost těchto definic vyplývá z citovaných zdrojů, práce se tímto nezabývá.

Klíčová slova: ohodnocené přechodové systémy, Hennessy-Milner logika, JavaScript, bisimulační ekvivalence, verifikační nástroj, bisimulační hry, grafický editor

Abstract

Verification of bisimulation equivalence of labeled transition systems is one of the standard techniques of system verification. The aim is to create a verification tool that checks this bisimulation equivalence on the inputs entered using a graphical editor and can also provide interactive playing of bisimulation games and evaluation of Hennessy-Milner logic formulas on the same inputs. This document describes the creation of this tool, including a descriptions and definitions of terms needed to understand the algorithms used. The correctness of these definitions results from the cited sources, this thesis does not include this.

Keywords: labeled transition system, Hennessy-Milner logic, JavaScript, bisimulation equivalence, verification tool, bisimulation games, graphical editor

Obsah

Seznam použitých zkratek a symbolů **3**

Seznam obrázků **4**

Seznam výpisů zdrojového kódu **5**

1 Úvod **6**
1.1 Cíle práce 6

2 Analýza problému **7**
2.1 Reaktivní systémy 7
2.2 Jazyk CCS 7
2.3 Ohodnocené přechodové systémy 9
2.4 Bisimulační ekvivalence 10
2.5 Bisimulační hry 11
2.6 Hennessy-Milner logika 13

3 Modelování **16**
3.1 Funkční analýza 16
3.2 Případy užití 17
3.3 Diagram tříd 17

4 Implementace **20**
4.1 Hlavní použité technologie 20
4.2 Grafická prezentace nástroje 24
4.3 Zadávání LTS grafem 25
4.4 Bisimulační ekvivalence a bisimulační hry 26
4.5 Vyhodnocování HML formulí 33
4.6 Persistence dat 41
4.7 Nasazení nástroje 42

5 Vstupy do programu **44**
5.1 Předem připravené vstupy 44
5.2 Importování souboru 44
5.3 Vytvoření vstupů v nástroji 46

6 Závěr **47**

Literatura **48**

Přílohy **49**

A Případy užití **50**

A.1 Zadání LTS grafem 50

A.2 Editace existujícího LTS 51

A.3 Zobrazení bisimilarity LTS 51

A.4 Vypočtení HML výrazů 52

B Seznam elektronických příloh **54**

Seznam použitých zkratek a symbolů

LTS	–	Ohodnocený přechodový systém
HML	–	Hennessy-Milner logika
JS	–	JavaScript jazyk
JSON	–	JavaScript objektová notace
HTML	–	Hyper Text Markup Language
DOM	–	Objektový model dokumentu
BOM	–	Objektový model prohlížeče
BNF	–	Backusova–Naurova forma
NPM	–	Node balíčkovací manager
HTTP	–	Hypertextový přenosový protokol
CCS	–	Milner kalkulus komunikujících systémů

Seznam obrázků

1 Rozhraní procesu CS 8
2 Grafy S a T pro silnou bisimulační hru 12
3 Diagram případů užití 17
4 Diagram tříd 18
5 Datové typy JavaScript jazyka 21
6 Hierarchie základních typů TypeScript jazyka 22
7 Příklad kvadratické Bézierovy křivky 25
8 Grafy S a U 27
9 Graf s τ akcí 29
10 Obrázek tabulky historie bisimulační hry 31
11 Diagram postupu vyhodnocení HML formulí 34
12 Ukázka převodu výrazů na tokeny 34
13 Ukázka převodu seznamu tokenů na AST 37
14 Ukázka vykreslení výsledku výpočtu formule 41

Seznam výpisů zdrojového kódu

1 Ukázka ReactJS komponenty s JSX 22
2 Ukázka ReactJS komponenty bez použití JSX 23
3 Příklad zadání kvadratické Bézierovy křivky v SVG 25
4 Pozicování textu akce na střed křivky 26
5 Pseudokód výherní strategie obránce 29
6 Pseudokód výherní strategie útočníka 32
7 Pravidla pro tokenizaci 35
8 Serializace souboru 41
9 Nahrání souboru 41
10 Konfigurace GitLab Pipeline pro nasazení na GitLab Pages 42
11 Příklad souboru k importu 45

1 Úvod

V této diplomové práci se budeme zabývat vytvořením nástroje do sady výukových komponent pro modelování a verifikaci systémů. Tento nástroj je určen k ověřování bisimulační ekvivalence na ohodnocených přechodových systémech a vyhodnocování formulí Hennessy-Milner logiky. V této kapitole jsou popsány cíle této práce.

V kapitole 2 jsou popsány a vysvětleny základní pojmy a zkratky potřebné k implementaci nástroje. Tyto pojmy se pak vyskytují v dalších kapitolách této práce.

Kapitola 3 se pak zabývá analýzou systému a modelováním jeho požadavků a prvků. Popisuje vztahy mezi rolí uživatele a systémem, dále popisuje kompletní možnosti systému a předpoklady pro jeho vývoj.

Kapitola 4 popisuje samotnou implementaci systému a algoritmy použité pro řešení jednotlivých částí. Nejprve jsou vysvětleny technologie použité pro vývoj nástroje a grafická reprezentace tohoto systému. Dále je popsán způsob zadávání grafů do sytému, výpočet ověření bisimulační ekvivalence a hraní her nad tímto výsledkem. Nakonec je popsáno vyhodnocování formulí Hennessy-Milner logiky a persistence dat v systému.

Kapitola 5 je zaměřena na zadávání vstupů do systému uživatelem a předpřipravenými daty pro vyzkoušení nástroje a importováním vstupních dat do systému.

V poslední kapitole 6 je uvedeno celkové zhodnocení systému a jeho přínos.

1.1 Cíle práce

Prvním cílem této práce je vytvoření dynamických webových stránek umožňující zadávat ohodnocené přechodové systémy grafem, zobrazit, zdali jsou dva ohodnocené přechodové systémy ekvivalentní podle bisimulační ekvivalence, a v kladném případě tuto ekvivalenci zobrazit. Dále zobrazit uživatelsky interaktivní popis vítězné strategie v bisimulační hře. Možnost zadat formule Hennessy-Milner logiky s rekurzí a zobrazit množiny stavů, ve kterých platí jednotlivé podformule a celá formule.

Druhý cíl je vytvořit ukázkové příklady tak, aby si uživatel mohl vše vyzkoušet i bez vlastních vstupů. Uživatel má mít také možnost uložit své vstupy do souboru a nahrát tyto soubory zpět do systému.

Třetím a posledním cílem této práce je nasazení webových stránek na veřejně dostupný server tak, aby uživatelé mohli tento systém používat ve webovém prohlížeči bez nutnosti spouštění stránek na vlastním serveru.

2 Analýza problému

V této sekci jsou vysvětleny základní pojmy, definice a jejich vlastnosti vyskytující se později v této práci. Hlavním tématem této sekce jsou ohodnocené přechodové systémy neboli LTS, bisimulační ekvivalence společně s hraním bisimulačních her na ohodnocených přechodových systémech a Hennessy-Milner Logika zkráceně HML.

2.1 Reaktivní systémy

V klasickém pohledu si výpočetní systémy na vyšší úrovni abstrakce můžeme představit jako černé skříňky, které přijímají vstupy a produkují odpovídající výstupy. Tento pohled také souhlasí s popisem algoritmických problémů. Algoritmický problém je specifikován kolekcí vstupů a pro každý vstup odpovídajícím výstupem. Pohled na výpočetní systém tudíž může být dán popisem, jak přeměnit vstupy na výstupy. Tato funkce může být částečná, tzn. může být nedefinovaná pro některé vstupy a nekončící.

V tomto pohledu je nekonečnost velmi nechtěná vlastnost. Algoritmus, který se neukončí pro některé vstupy, není takový, jaký bychom očekávali, že uživatel systému bude používat. Nicméně, existují systémy, které svou definicí jsou zamýšleny jako nekončící. Příkladem takových systémů jsou

- Operační systémy
- Komunikační protokoly
- Řídící systémy

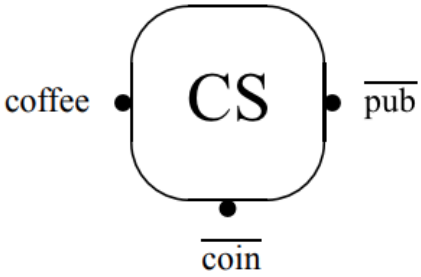
Tyto systémy fungují na základě výměny informací za běhu programu. Reaktivní systémy jsou ze své podstaty paralelní a jejich klíčové vlastnosti jsou komunikace a interakce s jejich výpočetním prostředím. Na standardní výpočetní systémy může být také nahlíženo jako na reaktivní systém, který interaguje se svým výpočetním prostředím pouze na začátku a konci svého výpočtu.

2.2 Jazyk CCS

CCS jazyk (zkratka z anglického Calculus of Communicating Systems) je kalkulus definující procesy a akce[1]. Tyto akce modelují nedělitelnou komunikace mezi dvěma účastníky. CCS je užitečný pro evaluaci vlastností jako jsou deadlock, livelock, starvation a podobně.

CCS proces si můžeme představit jako černou skříňku. Tato skříňka může mít jméno, které ji identifikuje a rozhraní procesu. Toto rozhraní popisuje kolekci komunikačních portů nebo-li kanálů. Tento proces může komunikovat s ostatními procesy ve stejném prostředí. Například v Obrázku 1 může komunikovat přes *coffee*, $\overline{coin}$ a $\overline{pub}$. Port *coffee* je použit pro vstup a porty $\overline{coin}$ a $\overline{pub}$ pro výstup. Obecně, máme-li port a, používáme $\overline{a}$ pro výstup na portu a.

Jak již bylo zmíněno, proces může mít jméno, stejně jako tomu je například u procedur. To znamená, že můžeme pojmenovat procesy a tato jména použít v definicích dalších procesů.



Obrázek 1: Rozhraní procesu CS

Jména procesů nám také umožňují definovat rekurzivní procesy stejně jako bychom definovali rekurzivní procedury ve standardních programovacích jazycích. $CM \stackrel{def}{=} coin.coffee.CM$ je stroj, který je ochoten vzít *coin* na vstupu, poté produkovat *coffee* na výstupu a vrátit se do původního stavu.

Volba je další CCS konstrukce, která nám nabízí možnost volby. CCS nabízí operátor volby $+$. Například stroj, který nabízí buďto $coffee$ nebo *tea* může být popsán následovně $CTM \stackrel{def}{=} coin.(\overline{coffee}.CTM + \overline{tea}.CTM)$, kde po obdržení *coin* na vstupu, proces CTM doručí buďto $coffee$ nebo *tea* podle uživatelovy volby a poté se vrátí do svého původního stavu.

Konstrukce paralelní kompozice je používána pro popis systémů běžících paralelně a v případě potřeby i komunikující mezi sebou. CCS nabízí operaci paralelní kompozice, kterou je možno zapsat jako $|$. Například CCS výraz $CM|CS$ popisuje systém obsahující dva procesy, CM a CS, které vůči sobě běží paralelně. Tyto dva procesy spolu mohou komunikovat přes komunikační porty, ovšem tak, že jeden z procesů využívá komunikační port jako vstup a druhý jako výstup.

Restrikce se snaží omezit rámec jmen kanálů, stejně jak tomu je u proměnných v blokově strukturovaných programovacích jazycích. Například, použitím operátorů *coin* a $coffee$ můžeme skrýt tyto porty v prostředí procesů CM a CS. Tento fakt můžeme zadefinovat následovně:

$$U \stackrel{def}{=} (CM|CS)coin.coffee \tag{1}$$

Přejmenování je operace, kterou můžeme použít pro generalizaci procesů. Takto můžeme zadefinovat generický proces, který například přijímá *coin* a vydává obecnou věc a poté se restartuje do počátečního stavu. Tento proces může vypadat takto: $VM \stackrel{def}{=} coin.\overline{item}.VM$. Pomocí přejmenování můžeme tento proces předělat na proces, který stále přijímá *coin*, ale místo obecné věci vrací *tea*: $TM \stackrel{def}{=} VM[tea/item]$. Zde $VM[tea/item]$ je proces, který se schová stejně jako VM, ale namísto generického *item* vydává *tea*.

Nakonec můžeme vše shrnout do této definice.

Definice: Mějme množinu jmen akcí, množina CCS procesů je pak definováno pomocí této BNF

gramatiky

$$P ::= \emptyset \quad | \quad A \quad | \quad a.P_1 \quad | \quad P_1 + P_2 \quad | \quad P_1|P_2 \quad | \quad P_1[L_1/L_2] \quad | \quad P_1\backslash L \tag{2}$$

Význam jednotlivých konstrukcí v definici je následující:

- $\emptyset$ - Prázdný proces.
- $a.P_1$ - Může vykonat akci a a pokračovat jako proces P_1.
- A - Identifikátor procesu.
- $P_1 + P_2$ - Volba procesu. Můžeme pokračovat jako proces P_1 nebo P_2.
- $P_1|P_2$ - Paralelní kompozice. Procesy P_1 a P_2 se vykonávají současně.
- $P_1[L_1/L_2]$ - Je proces P_1 se všemi akcemi z L_1 přejmenovanými na akce L_2.
- $P_1\backslash L$ - Je proces P_1 bez akcí z L.

2.3 Ohodnocené přechodové systémy

Ohodnocené přechodové systémy, zkráceně LTS (Labeled Transition System)[1], je jednoduchý systém pro modelování reaktivních, distribuovaných systémů založených na potencionálně nekonečných grafech s ohodnocenými hranami. Každý uzel grafu representuje stav v systému a hrany grafu představují přechody v systému označené akcí. Přechod $\rightarrow$ popisuje změnu ve stavech procesu: pokud proces s může vykonat akci a a tím se z něj stane proces s', napíšeme $s \xrightarrow{a} s'$. Někdy může být stav také označen jako počáteční[2] v rozšířené definici.

Definice: LTS jsou formálně definovány jako trojice $(S, A, \rightarrow)$, kde:

- S je neprázdná, potencionálně nekonečná, množina stavů
- A je konečná množina akcí
- $\rightarrow \subseteq S \times A \times S$ je přechodová relace

Notace: Mějme LTS $(S, A, \rightarrow)$, přechod $(s, a, s') \in \rightarrow$ je označen $s \xrightarrow{a} s'$, jenž se dá číst jako „systém ve stavu s může provést akci a do stavu s'". Zápis může být rozšířen na $v = a_1, a_2, a_3, a_4, a_5, ..., a_n$, kde $v \in A^*$, poté můžeme napsat $s \xrightarrow{v} s'$ právě tehdy, když existuje posloupnost stavů taková, že $s_0 = s$, $s_n = s'$ a $s_{(i-1)} \xrightarrow{a_i} s'_i$ pro každé i takové, že $1 \leq i \leq n$. Dále pak můžeme napsat $s \xrightarrow{*} s'$ pro nějaké takovéto v. Jinými slovy $\xrightarrow{*}$ je reflexivní a tranzitivní uzávěr relace $\rightarrow$.

Definice: [Dosažitelné stavy] Mějme $T = (Proc, Act, \{\xrightarrow{\alpha} \mid \alpha \in Act\})$, jakožto ohodnocený přechodový systém a mějme $s \in Proc$ jako počáteční stav. Řekněme, že $s' \in Proc$ je dosažitelný v systému T, tehdy a pouze tehdy, pokud $s \xrightarrow{*} s'$. Množina dosažitelných stavů obsahuje všechny stavy, které jsou v systému T dosažitelné z s.

2.4 Bisimulační ekvivalence

Bisimulace [1] je binární relace mezi stavy systému. Intuitivně jsou dva systémy bisimilární, pokud dokážou dorovnat své vzájemné tahy. Můžeme pak v tomto smyslu říct, že 2 systémy jsou bisimilární pokud vnější pozorovatel nedokáže tyto 2 systémy rozlišit. Bisimulační ekvivalence patří mezi ekvivalence chování (behavioral equivalences)[3], které mají společné to, že 2 ekvivalentní systémy nedovede pozorovatel rozlišit pouhým sledováním jejich chování. Různé typy ekvivalencí chování se od sebe vzájemně liší tím, jak velké rozdíly v chování systémů musí být, abychom již systémy za ekvivalentní nepovažovali. Bisimulační ekvivalence je jednou z nejznámějších a nejvýznamnějších z nich. Pojem bisimulace je někdy pozměněn přidáním dalších omezení či požadavků. Může se takto pojem bisimulace v různých kontextech nepatrně lišit. V ohodnocených přechodových systémech můžeme mít systém vybavený tichou akcí, běžně označovanou jako τ, to je akce, které je pro vnějšího pozorovatele neviditelná. Bisimulace pak může být oslabena definicí slabé bisimulace, ve které jsou tiché akce chápány odlišně od viditelných.

2.4.1 Silná Bisimilarita

Definice: Nechť $(S, A, \rightarrow)$ je LTS. Binární relace $R \subseteq S \times S$ je silná bisimulace právě tehdy, když pro každou dvojici stavů $(s, t) \in R$ a každou akci $a \in A$ jsou splněny následující podmínky:

- Jestliže existuje nějaké $s' \in S$ takové, že $s \xrightarrow{a} s'$, pak musí existovat nějaké $t' \in S$ takové, že $t \xrightarrow{a} t'$ a $(s', t') \in R$.
- Jestliže existuje nějaké $t' \in S$ takové, že $t \xrightarrow{a} t'$, pak musí existovat nějaké $s' \in S$ takové, že $s \xrightarrow{a} s'$ a $(s', t') \in R$.

Stavy s a s' jsou bisimulačně ekvivalentní neboli bisimilární $s \sim s'$, právě tehdy, pokud existuje nějaká bisimulace R taková, že $(s, t) \in R$. Relace $\sim$ se pak nazývá silná bisimulační ekvivalence neboli silná bisimilarita.

2.4.2 Slabá Bisimilarita

CCS může obsahovat vnitřní a pro vnějšího pozorovatele neviditelné akce, označované jako τ. Avšak i přes to, že jsou tyto akce neviditelné pro vnějšího pozorovatele, silná bisimilarita tento fakt nebere v úvahu a pracuje s těmito akcemi jako s jakýmikoliv normální akcemi. Proto je potřeba abstrahovat tyto vnitřní τ akce. Pro tuto abstrakci potřebujeme zavést novou notaci $\stackrel{\alpha}{\Rightarrow}$, dále pak můžeme pomocí této notace definovat slabou bisimilaritu[1].

Definice: Mějme CCS procesy P a Q, nebo obecně stavy v LTS. Pro každou akci α můžeme napsat $P \stackrel{\alpha}{\Rightarrow} Q$ tehdy a pouze tehdy, pokud platí jedno z následujících

- $\alpha \neq \tau$ a existují stavy P' a Q' takové, že $P(\xrightarrow{\tau})^* P' \xrightarrow{\alpha} Q'(\xrightarrow{\tau})^* Q$

- nebo $\alpha = \tau$ a $P(\xrightarrow{\tau})^* Q$,

kde píšeme $(\xrightarrow{\tau})^*$ pro reflexivní a tranzitivní uzávěr relace $\xrightarrow{\tau}$.

Definice: Slabá bisimulace je binární relace R nad množinou stavů LTS je bisimulace tehdy a pouze tehdy, pokud $s_1 \xrightarrow{\alpha} s_2$ a (α) je akce (zahrnující τ) taková že:

- pokud $s_1 \xrightarrow{\alpha} s_1'$, pak existuje přechod $s_2 \stackrel{\alpha}{\Rightarrow} s_2'$ takový, že $s_1' R s_2'$
- pokud $s_2 \xrightarrow{\alpha} s_2'$, pak existuje přechod $s_1 \stackrel{\alpha}{\Rightarrow} s_1'$ takový, že $s_1' R s_2'$

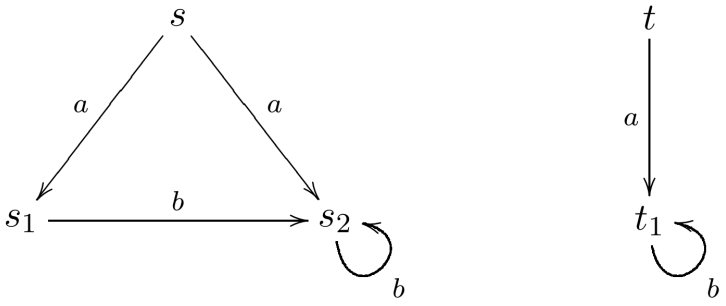
Tyto dva stavy s a s' jsou pozorovatelně ekvivalentní neboli slabě bisimilární, zapisováno jako $s \approx s'$ tehdy a pouze tehdy, pokud existuje slabá bisimulace, která se k nim vztahuje. Tudíž relace $\approx$ bude nazývána jako pozorovatelná ekvivalence nebo slabá bisimilarita.

2.5 Bisimulační hry

Jedna z technik, jak dokázat, že 2 stavy v systému jsou nebo nejsou bisimilární je pomocí bisimulačních her [1]. Myšlenka je, že máme dva hráče, kteří hrají bisimulační hru. Jeden z nich je nazýván útočník a druhý obránce. Útočník se snaží ukázat, že dva dvojice stavů není bisimulačně ekvivalentní a obránce se snaží ukázat opak. Formální definice je pak následovná.

Definice: Nechť $(Proc, Act, \{\alpha \rightarrow \mid \alpha \in Act\})$ je ohodnocený přechodový systém a mějme stavy s_1 a t_1, pro které chceme ověřit bisimulaci pomocí bisimulační hry. Silná bisimulační hra je hra dvou hráčů, útočníka a obránce. Hra je hraná v kolech a konfigurace hry jsou dvojice stavů z $Proc \times Proc$. V každém kole pouze jedna konfigurace hry je nazývána aktuální konfigurací. Na počátku je to (s_1, t_1). V každém kole má hráč šanci změnit aktuální konfiguraci (s, t) na základě následujících pravidel.

1. Útočník si vybírá buďto levou nebo pravou stranu aktuální konfigurace (s, t) a akci α z Act
   - Pokud útočník zvolí levou stranu konfigurace, pak musí provést přechod $s \xrightarrow{\alpha} s'$ pro nějaký stav s' z $Proc$.
   - Pokud útočník zvolí pravou stranu konfigurace, pak musí provést přechod $t \xrightarrow{\alpha} t'$ pro nějaký stav t' z $Proc$.
2. V tomto kroku musí obránce poskytnout odpověď na útok zahraný v předchozím kroku.
   - Pokud útočník zvolil levou stranu aktuální konfigurace, musí obránce zvolit pravou stranu aktuální konfigurace a odpovědět zahráním přechodu $t \xrightarrow{\alpha} t'$ pro nějaký stav t' z $Proc$.
   - Pokud útočník zvolil pravou stranu aktuální konfigurace, musí obránce zvolit levou stranu aktuální konfigurace a odpovědět zahráním přechodu $s \xrightarrow{\alpha} s'$ pro nějaký stav s' z $Proc$.



Obrázek 2: Grafy S a T pro silnou bisimulační hru

3. Konfigurace (s', t') se stane aktuální konfigurací a hra pokračuje dalším kolem podle pravidel.

Za odehraní jedné hry se považuje zahrání maximální možné sekvence konfigurací hráči podle pravidel popsaných výše a začínající v počáteční konfiguraci (s_1, t_1). Sekvence konfigurací je maximální pokud již nemůže být rozšířena hraním podle pravidel hry. Bisimulační hra může mít několik různých her odvozených od voleb tahů útočníka a obránce. Následně si můžeme popsat, kdy je hra vyhrána útočníkem, kdy obráncem, a jak se tyto výhry vztahují k bisimilaritě.

V konečné hře, hráč, který je zaseknutý a nemůže podle aktuálních pravidel hry provést přesun z aktuální konfigurace, prohraje hru. Za poznatek stojí, že útočník prohraje konečnou hru, pouze pokud obě $s \not\rightarrow$ a $t \not\rightarrow$, tj. Nedochází k přechodu jak z levé, tak z pravé strany konfigurace. Obránce prohraje konečnou hru, pokud nemá (na své straně konfigurace) žádný dostupný přechod odpovídající na akci vybranou útočníkem.

Může se také stát, že žádný z hráčů není zaseknutý v žádné konfiguraci a hra je nekonečná. V této situaci je obránce vítězem hry. Intuitivně je to přirozená volba výsledku, protože pokud je hra nekonečná, pak útočník nebyl schopen najít „rozdíl“ v chování obou systémů - což ukazuje bisimilaritu systémů.

Daná hra vždy vyhrává buď pro útočníka, nebo pro obránce. Nemůže být vítězná pro oba hráče současně.

Následující tvrzení se týká vztahu silné bisimilarity k odpovídající charakterizaci hry.

Tvrzení: Stavy s_1 a t_1 ohodnoceného přechodového systému jsou silně bisimilární, tehdy a pouze tehdy, má-li obránce univerzální výherní strategii v silné bisimulační hře počínaje konfigurací (s_1, t_1). Stavy s_1 a t_1 nejsou silně bisimilární, tehdy a pouze tehdy, pokud má útočník univerzální výherní strategii.

Univerzální výherní strategií máme na mysli, že hráč může vždy vyhrát hru, bez ohledu na to, jak si druhý hráč vybere své tahy. V případě, že má soupeř více než jednu volbu, jak pokračovat ze současné konfigurace, je třeba zvážit všechny tyto možnosti.

Pojem univerzální výherní strategie lze nejlépe vysvětlit pomocí příkladu.

Příklad: Ukážeme, že obránce má univerzální výherní strategii z konfigurace (s, t) na systémech S a T v Obrázku 2, a tedy ve světle předchozího tvrzení, že $s \sim t$. Abychom tak mohli učinit, musíme zvážit všechny možné pohyby útočníka z této konfigurace a definovat reakci obránce na každý z nich. Útočník může provést tři různé pohyby z (s, t).

1. Útočník vybere pravou stranu, akci a provede tah $t \xrightarrow{a} t_1$,
2. Útočník vybere levou stranu, akci a provede tah $s \xrightarrow{a} s_2$.
3. Útočník vybere levou stranu, akci a provede tah $s \xrightarrow{a} s_1$.

- Odpověď obránce na útok 1. je hraním $s \xrightarrow{a} s_2$. (Přestože existuje více možností, stačí poskytnout pouze jednu vhodnou odpověď.) Aktuální konfigurace se stane (s_2, t_1).
- Odpověď obránce na útok 2. je hraním $t \xrightarrow{a} t_1$. Aktuální konfigurace se opět stane $(s2, t1)$.
- Odpověď obránce na útok 3. je hraním $t \xrightarrow{a} t_1$. Aktuální konfigurace se stane $(s1, t1)$.

Nyní zbývá ukázat, že obránce má univerzální výherní strategii z konfigurací $(s2, t1)$ a $(s1, t1)$.

Z $(s2, t1)$ je snadno vidět, že jakékoliv pokračování hry bude vždy procházet stejnou aktuální konfigurací $(s2, t1)$, a proto bude hra nutně nekonečná. Podle definice vítězné hry je v tomto případě obráncem vítěz.

Z $(s1, t1)$ má útočník dva možné tahy. Buď $s_1 \xrightarrow{a} s_2$ nebo $t_1 \xrightarrow{a} t_1$. V prvním případě obránce odpoví $t_1 \xrightarrow{a} t_1$, v druhém případě $s_1 \xrightarrow{a} s_2$. Další konfigurace je v obou případech $(s2, t1)$ a my už víme, že obránce má z této konfigurace vítěznou strategii.

Ukázali jsme tedy, že obránce má univerzální výherní strategii z konfigurace (s, t) a podle tvrzení výše to znamená, že $s \sim t$. Výše uvedená charakterizace hry bisimilarity je jednoduchá, ale výkonná. Poskytuje intuitivní pochopení tohoto pojmu. Může být použita jak pro ukázání, že dva stavy jsou silně bisimilární, stejně tak dobře, že nejsou. Tato technika je zvláště užitečná pro ukázání, že dva stavy nejsou bisimilární.

2.6 Hennessy-Milner logika

Hennessy-Milner logika zkráceně HML je dynamická logika používaná k určení vlastností ohodnoceného přechodového systému (LTS). Rozšířená verze této logiky nazývaná HML s rekurzí dovoluje použití rekurze s využitím maximálních a minimálních pevných bodů.

2.6.1 Základní Hennessy-Milner logika

Formule je definována pomocí BNF gramatiky skrz set akcí Act ve vzorci 3 [1].

$$F, G ::= \; tt \;\mid\; ff \;\mid\; F \wedge G \;\mid\; F \vee G \;\mid\; \langle F \rangle \;\mid\; [a]F, \tag{3}$$

kde $a \in Act$, a kde používáme tt a ff pro značení $true$ a $false$. Pokud $A = \{a_1, ..., a_n\} \subseteq Act (n \geq 0)$, použijeme zkratku $\langle A \rangle$ pro formuli $\langle a_1 \rangle F \vee ... \vee \langle a_n \rangle F$ a $[A]F$ pro formuli $[a_1]F \wedge ... \wedge [a_n]F$. Pokud $A = \emptyset$, pak $\langle A \rangle F = ff$ a $[A] = tt$.

Co nás zajímá, je význam této logiky na CCS procesech a na stavech v LTS s akcemi Act. Toto může být slovně popsáno takto.

- Všechny procesy splňují tt.
- Žádné procesy nesplňují ff.
- Proces splňuje $F \wedge G$ (příslušně $G \wedge F$) tehdy a pouze tehdy, pokud splňuje oboje F a G (příslušně G a F).
- Proces splňuje $\langle a \rangle F$ pro nějaké $a \in Act$ tehdy a pouze tehdy, pokud může provést přechod akcí a do stavu splňujícího F.
- Proces splňuje $[a]F$ pro nějaké $a \in Act$ tehdy a pouze tehdy, pokud všechny jeho přechody akcí a, které z něj jsou provést, vedou do stavu splňujícího F.

Jinými slovy formule $\langle a \rangle F$ nám říká, že je možné vykonat akci a a tím splnit F a formule $[a]F$ nám říká, že nehledě na to, jak provedeme akci a, vždy skončíme ve stavu splňujícího F.

2.6.2 Tarskiho pevný bod

Teorém Tarskiho pevného bodu stanoví následovné [1, 4]:

Definice: Nechť L je úplný svaz a nechť $f : L \to L$ je monotónní funkce, pak množina pevných bodů z f v L je také úplným svazem. Element $d \in D$ je nazýván pevným bodem z f pokud a jen tehdy je-li $d = f(d)$.

Například funkce $f : 2^N \to 2^N$ definovaná pro každé $X \subseteq N$ jako $f(X) = X \cup \{1, 2\}$ je monotónní. Množina $\{1, 2\}$ je pevným bodem funkce f protože $f(\{1, 2\}) = \{1, 2\} \cup \{1, 2\} = \{1, 2\}$.

Teorém: Nechť $(L, \sqsubseteq)$ je úplný svaz a nechť $f : L \to L$ je monotónní funkce, pak f má největší fixní bod z_{max} a nejmenší fixní bod z_{min} daný jako

$$z_{max} = \sqcup \{x \in L | x \sqsubseteq f(x)\} \tag{4}$$

$$z_{min} = \sqcap \{x \in L | f(x) \sqsubseteq x\} \tag{5}$$

2.6.3 Hennessy-Milner logika s rekurzí

Invariant [1, 5] je matematická vlastnost objektu, která zůstává zachována po provedení operace nebo transformace určitého typu nad objektem. V našem případě invariant (Inv) znamená, že ať budeme provádět jakoukoliv akci formule F bude splněna. Například $Inv(\langle a \rangle tt)$ tak vyjadřuje, že ve všech dosažitelných stavech je uschopněna akce a. Naopak $Pos(F)$ znamená, že je možné dosáhnout stavu, kde platí formule F. Například $Pos([a]ff)$ znamená, že je možné dosáhnout stavu, kde akce a není proveditelná.

Nyní můžeme napsat rekurzivně definovanou formuli a říct, zdali chceme její nejmenší nebo největší řešení přidáním této informace ke nad znaménko rovná se. Pro $Inv(\langle a \rangle)$ chceme největší řešení, takže napíšeme

$$X \stackrel{max}{=} \langle a \rangle tt \wedge [Act]X \tag{6}$$

Pro $Pos([a])$ napíšeme

$$Y \stackrel{min}{=} [a]tt \vee \langle a \rangle Y \tag{7}$$

Obecně můžeme vyjádřit platnost formule F pro každý dosažitelný stav v LTS mající množinu akcí Act, psáno $Inv(F)$, rovnicí

$$X \stackrel{max}{=} F \wedge [Act]X, \tag{8}$$

a to, že F je splnitelné někdy v budoucnu, psáno $Pos(F)$, rovnicí

$$Y \stackrel{min}{=} F \vee \langle Act \rangle Y \tag{9}$$

3 Modelování

Tato sekce se zabývá specifikací nástroje, a to v podobě funkční analýzy, případů užití a doplňujících diagramů.

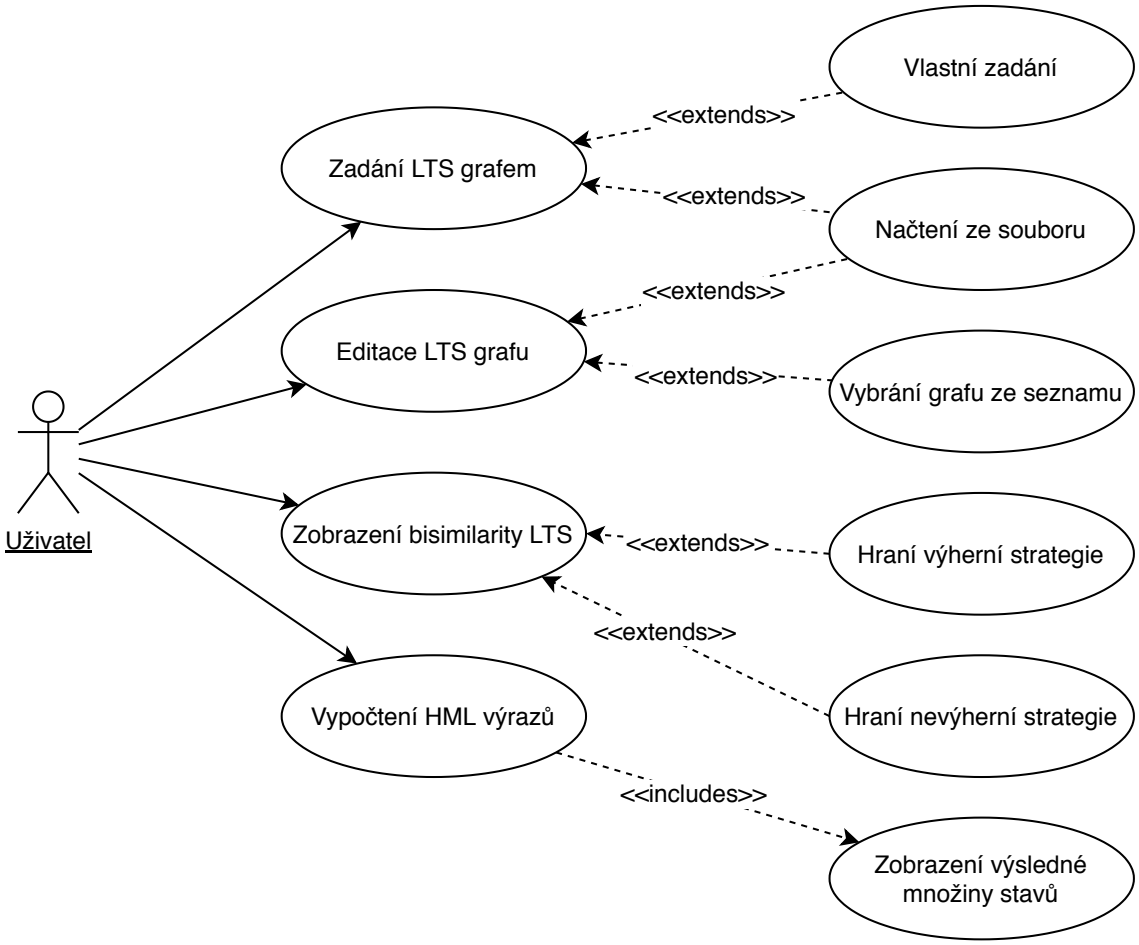
3.1 Funkční analýza

3.1.1 Funkční požadavky

- Systém musí umožnit uživateli zadat LTS pomocí grafu.
- Systém musí umožnit serializaci a deserializaci LTS do a ze souboru.
- Systém musí umožnit uložit uživatelem vytvořené LTS.
- Systém musí umět determinovat a zobrazit uživateli zdali jsou 2 LTS ekvivalentní podle bisimulační ekvivalence.
- Systém musí uživateli zobrazit interaktivní výherní strategii v případě, že 2 LTS jsou ekvivalentní podle bisimulační ekvivalence.
- Systém musí uživateli zobrazit interaktivní proherní strategie v případě, že 2 LTS nejsou ekvivalentní podle bisimulační ekvivalence.
- Systém musí uživateli umožnit zadat rekurzivní HML formuli a uložit ji pro pozdější použití.
- Systém musí umět zobrazit množiny stavů LTS, ve kterých platí jednotlivé podformule a celá formule.
- Systém musí obsahovat předpřipravené LTS grafy a HML formule, aby mohl být použit i bez uživatelem definovaných LTS grafů a HML formulí.

3.1.2 Nefunkční požadavky

- Systém musí být spustitelný ve webovém prohlížeči.
- Systémové GUI musí být srozumitelné pro cílenou skupinu uživatelů.
- Systém musí být dostatečně škálovatelný tak, aby dokázal poskytovat obsah všem uživatelům.
- Systém musí být spolehlivý.
- Systém musí být udržovatelný po dlouhou periodu času bez vnějšího zásahu.
- Systém musí být rozšiřitelný.
- Systém musí být schopen automatického zotavení v případě systémové chyby.



Obrázek 3: Diagram případů užití

3.2 Případy užití

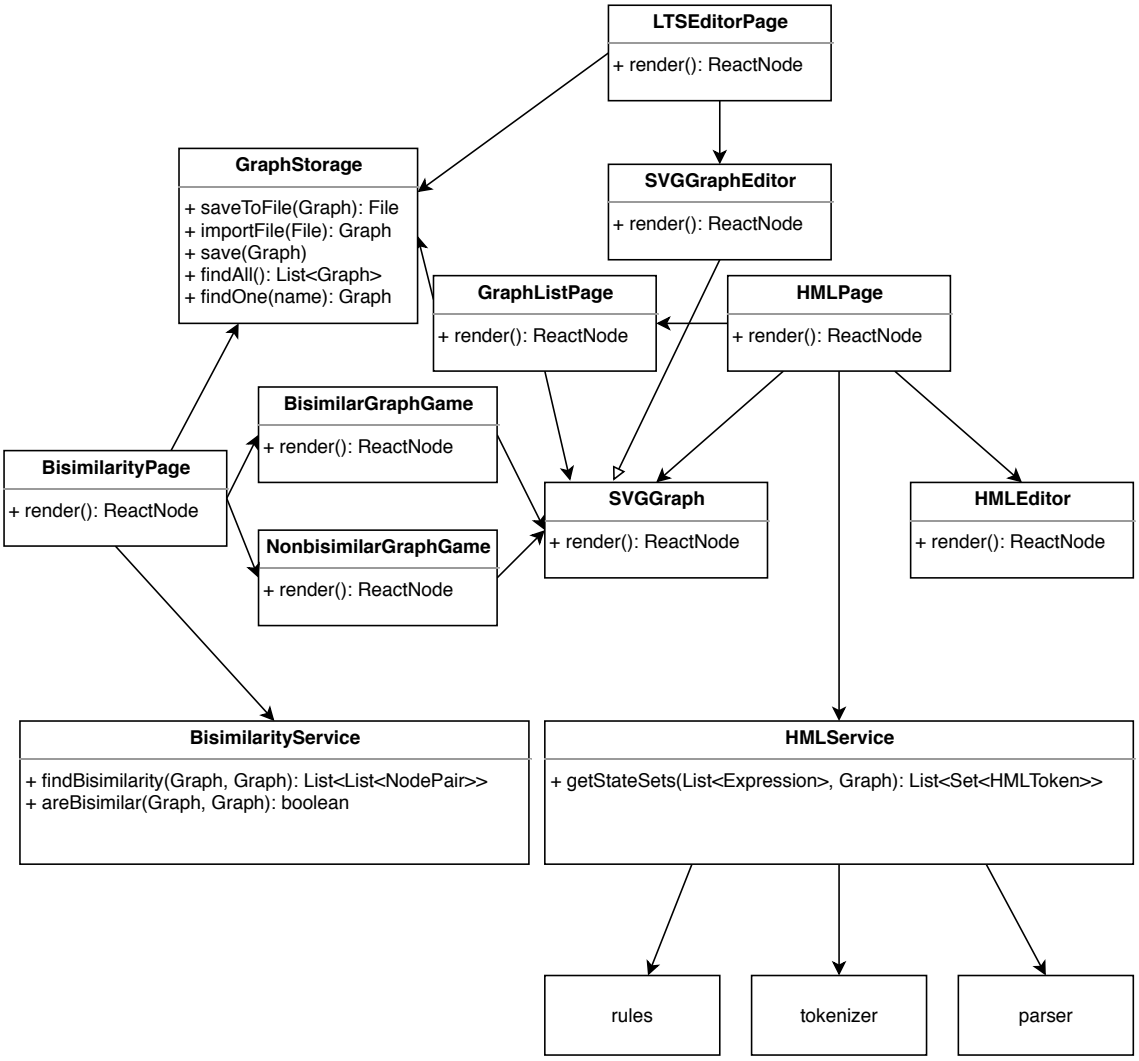
V diagramu případů užití 3 jsou popsány hlavní scénáře využívání nástroje uživatelem. Tyto hlavní případy zahrnují následující možnosti:

- Zadání LTS grafem
- Editace LTS grafu
- Zobrazení bisimilarity LTS
- Vypočtení HML výrazů

Tyto scénáře jsou plně rozepsány v příloze A.

3.3 Diagram tříd

V diagramu tříd 4 vidíme základní strukturu tříd a funkcí v implementaci nástroje. Ten se skládá z několika jednotlivých komponent typu stránky (anglicky Page), které se starají o logiku



Obrázek 4: Diagram tříd

vykreslení jednotlivé stránky, poté z dalších komponent řešících vykreslení a logiku spjatou s výpočty a vykreslením určitých požadavků na stránkách, například SVGGraph komponenta, která se stará o logiku vytváření a vykreslení LTS grafů. Nakonec obrázek obsahuje několik tříd typu služba (anglicky Service), které se starají o hlavní výpočetní logiku aplikace. Mnoho menších komponent je pro přehlednost z obrázku vypuštěno, popis komponent na obrázku 4 je v následujícím seznamu.

- LTSEditorPage - Stránka zodpovědná za vykreslování vytváření a editaci LTS grafu.
- GraphListPage - Stránka zodpovědná za vykreslování uložených grafů.
- HMLPage - Stránka zodpovědná za vykreslování HML komponent.
- BisimilarityPage - Stránka zodpovědná za vykreslování bisimulačních her.

- SVGGraphEditor - Editor umožňující tvorbu a editaci LTS grafů.
- SVGGraph - Komponenta zobrazující LTS graf.
- HMLEditor - Editor pro vytváření a ověřování HML formůlí.
- BisimilarGraphGame - Bisimulační hra na dvou bisimulačně ekvivalentních grafech.
- NonbisimilarGraphGame - Bisimulační hra na dvou bisimulačně neekvivalentních grafech
- BisimilarityService - Logika zodpovědná za ověřování bisimilarity.
- GraphStorage - Logika ukládání a načítání grafů do a ze souborů a persistentní paměti prohlížeče.
- HMLService - Logika zodpovědná za HML.
- rules - Definice pravidel pro vytváření tokenů.
- tokenizer - Převod formule na tokeny.
- parser - Převod tokenů na abstraktní syntaktický strom.

Hlavní části komponent a jejich logiky jsou dále rozepsány a vysvětleny v následujících kapitolách.

4 Implementace

V této sekci se budeme zabývat popisem konkrétní implementace nástroje pro zadávání LTS grafem, ověřování bisimulační ekvivalence, výpočtem strategií potřebných k hraní silných bisimulačních her, a vyhodnocování HML výrazů s i bez rekurze. Rozebereme si jak technologie použité pro docílení zadání, tak funkčnost jednotlivých částí. Probereme také algoritmy, které mohou být použity k dosažení této funkčnosti a algoritmy použité pro dosáhnutí funkčnosti v konkrétním případě.

Celý nástroj je navržen tak, aby jeho výpočty byly prováděny na hardware uživatele a nevytvářely tak zátěž na centrální server. Tímto eliminujeme potřebu pro silný centrální server a uživatel tak není omezen zatížením tohoto centrálního serveru, ale jen výpočetními prostředky jeho vlastního systému. Pro docílení byly použity webové známé technologie jako jsou HTML, CSS a JavaScript. Dále bylo použito několik knihoven pro dosažení vysoké kvality, lepší udržitelnosti, a snadnějšího vývoje nástroje. Příkladem je TypeScript a ReactJS. Základní informace o nejen těchto technologiích jsou popsány v následujících kapitolách. Tyto zvolené technologie jsou známé a populární v prostředí vývoje v JavaScript jazyce, pro hlubší porozumění jakékoliv technologie je tedy možno navštívit stránky dokumentace této technologie.

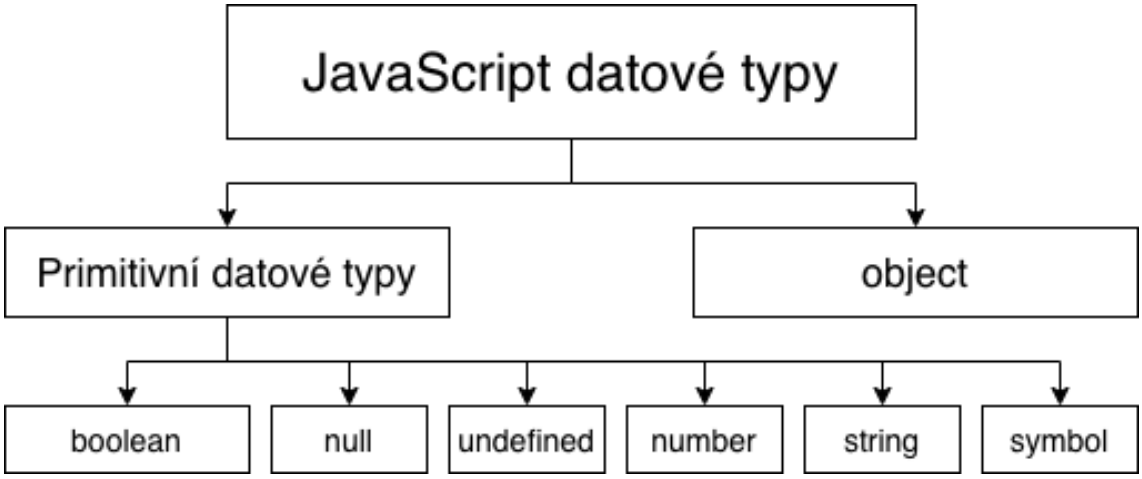
4.1 Hlavní použité technologie

Jedním z nefunkčních požadavků je nenáročnost na serverové zdroje, proto je nástroj vyvinut tak, aby výpočty dělal na klientské straně. Nástroj je vyvinut jako webová aplikace, která je uživateli doručena v podobě HTML, CSS a JavaScript. Pro udržitelnější vývoj je ale použito několik knihoven a jazyků. Hlavní použitá knihovna pro vývoj nástroje je ReactJS a jako programovací jazyk je použit převážně TypeScript. Dále je použit Babel pro překlad TypeScript jazyka do JavaScript a Webpack pro balíčkování aplikace pro klienty. Údržba závislostí projektu je zařízena pomocí NPM.

4.1.1 ECMAScript a JavaScript

ECMAScript[6, 7] je specifikace skriptovacího jazyka a byl vytvořen pro standardizaci JavaScript jazyka. JavaScript je nejpoužívanější implementací tohoto ECMAScript standardu, ovšem i další implementace existují, například JScript a ActionScript. ECMAScript je používán pro skriptování klientských webových aplikací, backendových služeb pomocí NodeJS, desktopových i mobilních aplikací.

EcmaScript byl poprvé představen v roce 1997 ve verzi 1, o rok později ve verzi 2 a v roce 1999 ve verzi 3. Následující vydaná verze byla 5 v roce 2009 a 5.1 v roce 2011. Tyto nové verze přinesly mnoho nových specifikací ve standardu ovšem za cenu dlouhé prodlevy mezi verzemi. V roce 2015 ECMAScript přešel na nový způsob verzování začínající ES a pokračující číslem roku, například ES2015. Nová verze vychází každý rok s malou sadou nových specifikací.



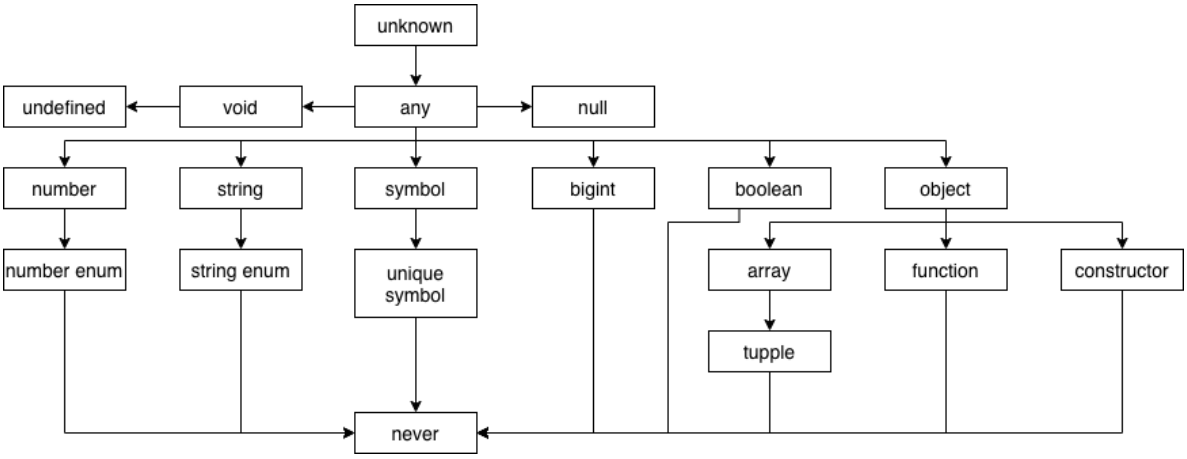
Obrázek 5: Datové typy JavaScript jazyka

Jak již bylo řečeno, JavaScript je implementací ECMAScript specifikace, ovšem JavaScript přidává do jazyka dalších API (zkratka pro Application Programming Interface) jako je například: Document Object Model (DOM) a Browser Object Model (BOM). JavaScript je just-in-time kompilovaný, dynamicky typovaný jazyk obsahující objektově orientované konstrukce založené na prototypech a obsahuje takzvané first-class funkce. Typy jsou asociované s hodnotou, raději než s výrazem, například proměnná může být vázaná na typ number a později přeřazena na typ string. Jazyk také nabízí funkci eval jenž může vykonat JavaScript kód dodaný jako řetězec textu za běhu programu.

4.1.2 TypeScript

TypeScript[8] je nadmnožinou JavaScript jazyka, tzn. jakýkoliv JavaScript kód je validním TypeScript kódem. TypeScript přidává možnost statických definicí typů a typovou kontrolu. Přináší také základní hierarchii statických typů a jejich dědičností viditelnou v obrázku 6. JavaScript nabízí základní datové typy za běhu programu, ale chybí mu statická typovost a proto vznikl TypeScript jako jedna z možností, jak tuto funkcionalitu přidat.

Statické typy poskytují hlavně lepší dokumentaci a tím i udržitelnost kódu a umožňují typovou validaci kódu a potencionální objevení chyb v kompilačním čase namísto za běhu programu. Psaní typů v TypeScript jazyce je dobrovolné, TypeScript dovede mnoho typů odvodit i bez jejich explicitní definice. Pro spuštění TypeScript jazyka je ale potřeba tento kód převést do JavaScript jazyka, a to lze například za pomoci TypeScript překladače a nebo za pomoci knihoven jako je Babel. Statické typy jsou v kompilačním čase odstraněny a výsledkem je čistý JavaScript kód.



Obrázek 6: Hierarchie základních typů TypeScript jazyka

4.1.3 ReactJS

ReactJS[9] je JavaScript knihovna pro tvorbu uživatelských rozhraní. Cílem ReactJS je psát deklarativní komponenty, které se efektivně aktualizují a vykreslují, když se změní jejich stav. ReactJS se zabývá hlavně efektivním vykreslováním dat do Document Object Model (DOM) prohlížeče. Pro dosáhnutí tohoto cíle, ReactJS má vlastní kopii DOM zvanou VirtualDOM. Všechno vykreslování je tedy provedeno do virtuálního DOM a ReactJS poté pomocí algoritmu vyhodnotí rozdíl mezi virtuálním DOM a reálným DOM a provede co možná nejmenší počet operací nad DOM pro převedení stavu z virtuálního DOM do reálného DOM.

ReactJS přidává vlastní XML nadstavbu do JavaScript jazyka s jménem JSX. JSX je pouze syntaktická nadstavba JavaScript jazyka představená společně s ReactJS. JSX je velmi podobná HTML a díky tomu má mírnou učící křivku. Dále také zabraňuje takzvanému HTML injection útoku (XSS). JSX syntaxe avšak nepatří do standartu ECMAScript a tím pádem není zpracovatelný žádným JavaScript standartním prohlížečem. JSX musí být před spuštěním aplikace přeloženo do standardů, které nabízí ECMAScript. Následující 2 příklady jsou ekvivalentní, první příklad používá JSX, druhý příklad je ukázka zápisu bez JSX. Transpilátor pak tedy musí transformovat kód prvního příkladu na kód druhého příkladu.

```
class HelloMessage extends React.Component {
  render() {
    return (
      <h1 className="greeting">
        Hello {this.props.name}
      </h1>
    )
  }
}
```

Výpis 1: Ukázka ReactJS komponenty s JSX

```
class HelloMessage extends React.Component {
  render() {
    return (
      React.createElement(
        'h1',
        {className: 'greeting'},
        'Hello' + this.props.name
      )
    )
  }
}
```

Výpis 2: Ukázka ReactJS komponenty bez použití JSX

ReactJS komponenty jsou strukturovaný podobně jako DOM, tj. jako strom. ReactJS pak poskytuje několik možností k manipulaci dat mezi komponentami v tomto stromě. Prvním z nich je proměnná *props*. *props* si můžeme představit podobně jako vstupní argument do funkce. Komponenta může svým potomkům poslat data přes tyto *props*, ve chvíli kdy potomek zjistí změnu těchto dat, automaticky se překreslí s novými daty.

Druhou možností je vnitřní stav komponenty (*state*), který je navenek neviditelný pro ostatní komponenty. Tento stav můžeme uvnitř komponenty měnit za pomocí funkce *setState*, kdy při každém zavoláním *setState* se detekuje změna a komponenta se překreslí, všichni potomci této komponenty pak také dostanou povel k překreslení.

Často při vývoji nastává požadavek na změnu dat v rodiči vyvolanou některým z potomků. V případě, že se jedná o přímého potomka, můžeme v rodiči vytvořit funkci, která pracuje se *setState* a tuto funkce poslat potomkovi jako *props*. Potomek pak může tuto funkci zavolat, případně jí i předat data v podobě parametrů funkce a tím v podstatě zavolat *setState* rodiče. Výsledkem je pak změna *state* a překreslení jak rodiče tak jeho potomků. Toto je v ReactJS velmi často vyskytující se vzor. Předávání funkce *setState* potomkům přímo není doporučeno a je označováno jako špatný návrhový vzor.

Pokud chceme předat data nebo funkce z jedné komponenty do komponenty o několik stupňů níže ve struktuře, může být postupné předávání dat pomocí *props* zbytečně se opakující. Například komponenta A předá proměnnou do komponenty B, ta pokračuje s předáváním této proměnné do komponenty C, ta pokračuje s předáním do komponenty D. V tomto případě by bylo lepší, pokud by si komponenta D mohla požádat o data přímo z komponenty A. ReactJS k tomuto účelu nabízí tzv. Context. Můžeme tak vytvořit Context, který obsahuje poskytovatele dat, tzv. Provider, jehož účelem je držet a poskytovat data. V Aplikaci pak můžeme použít

i několik instancí tohoto vytvořeného poskytovatele, kde každý z nich má vlastní stav. V našem případě tak vyjmeme stav z komponenty *A* a přeneseme ho Provider komponenty. Jakákoliv komponenta, která je ve stromě pod tímto poskytovatelem, může použít jeho Context a zaregistrovat se k němu jako konzument dat, tzv. Consumer. Vyhledávání správného poskytovatele ke konzumentovi, je prováděno knihovnou ReactJS. A to tak, že ReactJS prohledává strom komponent, od konzumenta směrem nahoru, do doby než narazí na poskytovatele dat, z kterého si data vezme.

4.1.4 Babel a Webpack

Babel[10] je JavaScript transpilační kompilátor neboli transpilátor (anglicky transpiler) rozširitelný pomocí takzvaných pluginů. Transpilátor je typ překladače, který bere na vstupu zdrojový kód napsaný v programovacím jazyce a produkuje ekvivalentní zdrojový kód ve stejném nebo jiném programovacím jazyce. Transpilátor konvertuje mezi programovacími jazyky na stejné nebo podobné úrovni abstrakce, kdežto kompilátor (anglicky compiler) překládá z programovacího jazyka vyšší abstrakce do programovacího jazyka nižší abstrakce. Avšak tyto dva pojmy jsou v dokumentaci a článcích často zaměňovány a jejich rozdíl je přehlížen, viz například TypeScript kompilátor[8].

Webpack[11] je nástroj pro budování statických modulů kódu pro moderní JavaScript aplikace. Když Webpack z pracovává kód aplikace, vnitřně si vytvoří závislostní graf, který mapuje každý modul aplikace a jejich závislostí, a vygeneruje jeden nebo více výstupních JavaScript balíčků. Toto nám umožní uživateli servírovat jen jeden nebo pár JavaScript souborů namísto stovek až tisíců souborů a knihoven.

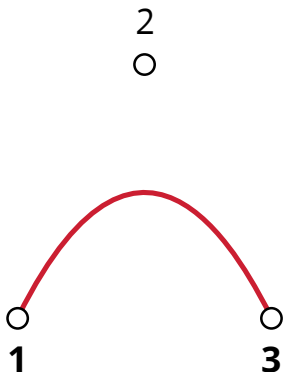
Webpack[11] také poskytuje tzv. tree shaking algoritmus pro eliminaci mrtvého kódu. Oproti jiným technikám eliminace mrtvého kódu, tree shaking začíná z vstupního místa programu a zahrnuje do výsledného balíku jen kód, u kterého je garantováno, že bude vykonán. Může tedy být popsán jako zahrnování živého kódu.

4.1.5 NPM

Node Package Manager (NPM)[12] je nejpoužívanější JavaScript balíčkovací systém pro distribuci knihoven a světově největší poskytovatel softwarových knihoven. NPM používá pro definici projektu a závislostí na knihovnách soubor package.json, který musí být zapsán ve formátu JSON. Tyto definované závislosti pak mohou být vývojářem nainstalovány pomocí jednoho příkazu. Soubor package.json také slouží k definici jména projektu, autora, licence a dalších užitečných informací pro publikaci vlastní knihovny.

4.2 Grafická prezentace nástroje

Grafické rozložení nástroje se skládá z menu, hlavičky a hlavního obsahu. Menu zůstává po dobu používání nástroje statické a slouží pouze pro hlavní navigaci v nástroji. Hlavička obsahuje



Obrázek 7: Příklad kvadratické Bézierovy křivky

informace o tom, kde v nástroji se uživatel nachází. Obsah jako takový je hlavním interaktivním prvkem nástroje a je odlišný pro každou položku v menu a to:

- Hlavní přehled obsahující informace o nástroji.
- Zadávání LTS grafem.
- Seznam uložených grafů.
- Ověřování a hraní bisimulačních her.
- Tvorba a ověřování HML formulí.

4.3 Zadávání LTS grafem

Pro zadávání LTS je v nástroji vytvořen editor grafů, který umožňuje přidávat, pojmenovávat a pozicovat uzly grafu, propojovat uzly grafu hranami a zapisovat na hranu jednu nebo více akcí odděleny znakem čárky a pozicovat linku hrany grafu pomocí kvadratické Bézierovy křivky. Editor grafů používá k vykreslení webový standard SVG (škálovatelná vektorová grafika). Pro vykreslení linek hran pomocí kvadratické Bézierovy křivky jsou potřeba 3 bodové souřadnice, a to počáteční bod P_0, koncový bod P_2 a kontrolní bod P_1.

Kvadratická Bézierova křivka je cesta vyznačená funkcí B(t), máme-li body P_0, P_1 a P_2.

$$\mathbf{B}(t) = (1-t)[(1-t)\mathbf{P}_0 + t\mathbf{P}_1] + t[(1-t)\mathbf{P}_1 + t\mathbf{P}_2] \ , \ 0 \leq t \leq 1, \tag{10}$$

Pomocí SVG tuto křivku pak můžeme zadat například takto:

```
<path d="M100,250 Q250,100 400,250" />,
```

Výpis 3: Příklad zadání kvadratické Bézierovy křivky v SVG

kde obecný příkaz M určuje přesunutí na pozici x a y, v našem případě reprezentuje bod P_0. Příkaz Q pak reprezentuje zbylé souřadnice kvadratické Bézierovy křivky, a to jsou v našem případě P_1 a P_2.

Dále potřebujeme vyřešit problém pozicování textu akce na středu křivky. Na tento problém je použit kód 4.

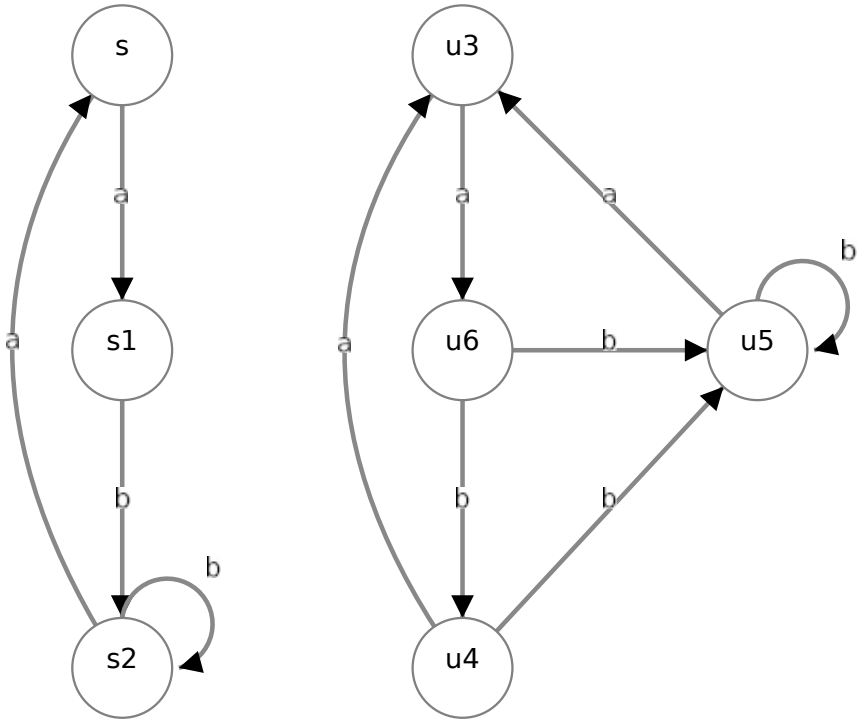
```
/**
 * výpočet relativní pozice bodu na křivce za použití t a c
 *
 * @param t je číslo mezi 0 a 1 určující, kde chceme bod,
 *   např. pro bod uprostřed použijeme t=0.5
 * @param c je pole 3 bodů:
 *   1. počáteční bod křivky (vždy (0,0))
 *   2. "pozicovací" bod křivky
 *   3. konečný bod křivky
 * @returns {{x: number; y: number}}
 */
function quadraticBezierCurvePoint(t, c) {
  const t1 = 1 - t
  const t1_2a = t1 * t1
  const t1_2b = 2 * t * t1
  const t1_2c = t * t
  return {
    x: c[0].x * t1_2a + t1_2b * c[1].x + t1_2c * c[2].x,
    y: c[0].y * t1_2a + t1_2b * c[1].y + t1_2c * c[2].y,
  }
}
```

Výpis 4: Pozicování textu akce na střed křivky

4.4 Bisimulační ekvivalence a bisimulační hry

Algoritmus použit pro ověření bisimulační ekvivalence je založen na postupné eliminaci dvojic stavů. Tento algoritmus, má horší složitost než jiné, v literatuře popsané, algoritmy, ale jeho implementace je značně jednodušší. Cílem této práce není použití nejefektivnějšího algoritmu ale vypočtení relativně malých bisimulačních ekvivalencí. Mějme 2 obecné LTS grafy G1 a G2. V našem algoritmu začneme s množinou dvojic všech uzlů z grafu G1 s všemi uzly v grafu G2. Nyní můžeme iterovat nad touto množinou a postupně eliminovat dvojice uzlů, které nepatří do potencionální bisimulační ekvivalence.

Jako první vyřadíme dvojice uzlů, které nemají stejné akce výchozích hran. Poté vyřadíme dvojice uzlů, do kterých nevedou hrany stejných akcí, a nakonec vyřazujeme dvojice, do kte-



Obrázek 8: Grafy S a U

rých se z aktuální omezené množiny nedá přes hrany dostat do doby, než dostaneme stabilní množinu páru stavů grafu, která se již nemění. Tato množina může být i prázdná. Aby byly 2 systémy bisimulačně ekvivalentní, musí se všechny stavy obou systémů v této množině nacházet. Algoritmus tento fakt ověří. Tímto postupem dostaneme množinu dvojic stavů odpovídající silné bisimilaritě, pokud jsou systémy silně bisimulační. Tuto množinu poté vypíšeme uživateli a dále také využijeme v algoritmech pro hraní silně bisimulačních her. Pro hraní her na bisimulačně ekvivalentních systémech si vystačíme s touto množinou stavů, nicméně pro hraní her na systémech, které nejsou bisimulačně ekvivalentní, budeme muset tento eliminační algoritmus rozšířit. Toto rozšíření si popíšeme v kapitole zabývající se hraním her na systéme které nejsou bisimulačně ekvivalentní.

Příklad: Mějme 2 LTS grafy znázorněné na Obrázku 8. Množina stavů *Proc* grafu S je dána stavy $\{s, s1, s2\}$ a stavů *Proc* grafu U je dána stavy $\{u, u1, u2, u3\}$. Množina kombinace stavů obou grafů je tedy $\{(s,u),(s,u1),(s,u2),(s,u3),(s1,u),(s1,u1),(s1,u2),(s1,u3),(s2,u),(s2,u1),(s2,u2),(s2,u3)\}$. V prvním kroku odstraníme všechny dvojice stavů, které nemají stejné akce na výstupních hranách. Odstraníme tedy $\{(s,u1),(s,u2),(s,u3),(s1,u),(s1,u2),(s1,u3),(s2,u),(s2,u1)\}$, tím nám zbude množina $\{(s,u),(s1,u1),(s2,u2),(s2,u3)\}$. Dále ověříme do kterých párů stavů vedou stejné akce, těmi jsou $\{(s,u),(s1,u1),$

$(s2, u2), (s2, u3)\}$. Nyní můžeme ověřit, zda všechny akce z této množiny vedou opět do stavů v této množině. (s, u) vede přes akci a do $(s1, u1)$, která do naší množiny spadá, tudíž tento pár stavů v množině ponecháme. $(s1, u1)$ vede přes akci b do $(s2, u2)$ a $(s2, u3)$, obě dvojice spadají do naší potencionální bisimulační množiny, takže je opět ponecháme v množině. $(s2, u2)$ vede přes akci a do (s, u) a přes akci b do $(s2, u3)$, obě dvojice spadají do naší potencionální bisimulační množiny, takže je opět ponecháme v množině. Posledním prvkem k otestování je $(s2, u3)$, z těchto stavů se můžeme přes akci a dostat do (s, u) a přes akci b do $(s2, u3)$, obojí pak patří do naší množiny, tudíž je opět ponecháme. Naše množina nyní vypadá takto $\{(s, u), (s1, u1), (s2, u2), (s2, u3)\}$ a je shodná s množinou, u které jsme toto ověřování začínali, tudíž je to naše bisimulační množina.

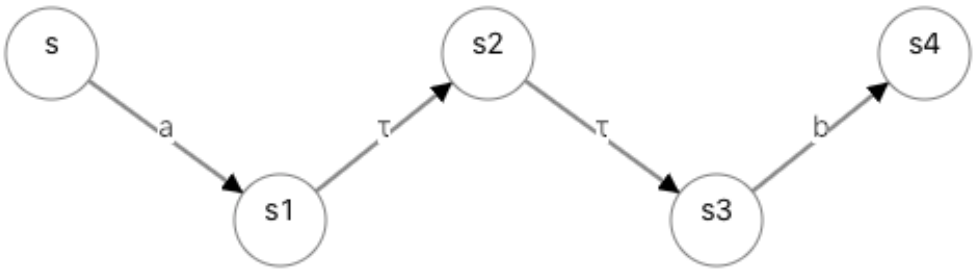
Tento postup pomocí eliminace se neřadí mezi nejefektivnější způsoby řešení, avšak jeho implementace je poměrně jednoduchá. Díky tomu, že nepočítáme, že by se v systému používaly LTS grafy s velkým počtem stavů, je tento algoritmus pro naše použití dostačující. Existují však i efektivnější algoritmy zmíněné například v článcích [13] a [14], v této práci se ovšem těmito algoritmy zabývat nebudeme.

4.4.1 Slabá bisimulační ekvivalence

Jedním z možných rozšíření nástroje je výpočet slabé bisimulační ekvivalence a hraní slabých bisimulačních her, pro které tuto slabou bisimulaci potřebujeme. Dále potřebujeme u tahů ve hře s tuto slabou bisimulační ekvivalencí a s neviditelnými akcemi τ počítat. Pro představu mějme graf 9. Zde jako útočník můžeme ze stavu s jít přes hranu a do jakéhokoliv ze stavů $\{s1, s2, s3\}$, stejně tak pokud bychom měli bránit takovýto tah. Musíme tedy v našich algoritmech počítat s těmito tahy. Dalším případem je pak přechod z například $s2$ do $s4$ zahráním hrany b, tento tah zahrnuje přechod pomocí akce τ a až poté pomocí akce b. Obecně budeme potřebovat kontrolovat tyto dva případy, a to pokud je možno přejít přes požadovanou hranu do nového stavu, je potřeba se podívat jestli z tohoto stavu vedou nějaké hrany τ do dalších stavů a pro tyto stavy tento fakt rekurzivně ověřit, všechny tyto stavy poté budou použity pro jak nabídku tahu uživateli, tak výběr tahu algoritmu. V druhém případě začínáme ve stavu, který již hranu τ nabízí, a tudíž je potřeba zkontrolovat, zdali stav, do kterého tato hrana vede, nebo další stavy dosažitelné z původního stavu pomocí přechodů τ, obsahují možnost zahrání požadované akce b.

4.4.2 Výherní strategie obránce

V případě, že jsou 2 grafy vybrané uživatelem bisimulačně ekvivalentní, umožníme uživateli hrát bisimulační hru. Uživatel bude v tomto případě hrát z pozice útočníka a nástroj, v pozici obránce, bude úspěšně bránit jakékoliv uživatelovy útoky. Hra může skončit buďto nemožností uživatele zahrát jakýkoliv útok nebo dostáním se do konfigurace ve které se hra již nacházela s nemožností hraním historicky odlišné akce. Správně by hra měla pokračovat v cyklu do nekonečna a tím



Obrázek 9: Graf s τ akcí

bychom dokázali, že tyto 2 systémy jsou bisimulačně ekvivalentní, toto ovšem není uživatelsky přívětivý způsob.

Bisimulační hra je hrána na kola, kde každé kolo je sestaveno z fází:

1. Útočník si vybere graf, na kterém chce vykonat tah.
2. Útočník provede tah z aktuální konfigurace přes hranu do nové konfigurace.
3. Obránce hraje zahraje odpovídající tah z aktuální konfigurace druhého grafu.

V našem případě může hra skončit 2 způsoby, a to:

1. Útočník nemůže z aktuální konfigurace hrát na žádném grafu.
2. Útočník vyzkoušel všechny možné konfigurace hry a další tahy by vedly do konfigurací, ve kterých se hra již v minulosti nacházela.

```
let eq = // pole dvojic stavů bisimulační ekvivalence

let gameHistory = [/* počáteční stavy */] // pole historie tahů hry
while(true) {
    let actualConfiguration = gameHistory.last()
    let canAttack = zjistíme zdali může uživatel někam táhnout z
        actualConfiguration
    if(!canAttack) {
      return // konec hry
    }
    let possibleAttackMoves = vypočítání a nabídnutí možných tahů z
        actualConfiguration uživateli
    let attackMove = počkáme na tah uživatele na grafu ze stavu 1 pomocí akce
        do stavu 2
```

```
    let defendMove = najdeme konfiguraci do které se uživatel může dostat přes
        stejnou hranu
    hameHistory.push({
        attackMove,
        defendMove,
    })
}
```

Výpis 5: Pseudokód výherní strategie obránce

Z toho nám vyplývá, že jako obránce potřebujeme algoritmus, který pro jakýkoliv útočný tah uživatele dovede odpovědět tahem takovým, aby algoritmus nikdy neprohrál. Mějme graf G daný stavy $\{g_1, g_2, g_3, g_4\}$ a přechody $g_1 \xrightarrow{a} g_2, g_2 \xrightarrow{a} g_3, g_2 \xrightarrow{b} g_4$. Dále mějme hru nad dvojicí grafů (G, G). Tato dvojice grafů je bisimulačně ekvivalentní a její bisimulační ekvivalence je následovná $\{(g_1, g_1), (g_2, g_2), (g_3, g_3), (g_4, g_4)\}$. Cílem našeho algoritmu bude reagovat na tahy útočníka takovým způsobem, že se budeme vždy táhnout tak, abychom se dostali do konfigurace hry, která odpovídá některé dvojici z naší bisimulační ekvivalence. Například pokud je aktuální konfigurace hry $(g2, g2)$ a útočník zahraje hranu $g_2 \xrightarrow{b} g_4$, vznikne nám konfigurace $(g4, g2)$. Jediná dvojice stavů v bisimulační ekvivalenci, která má na levé straně g_4 a můžeme se do této konfigurace dostat zahráním stejné akce jako útočník, je (g_4, g_4). Po zahrání $g_2 \xrightarrow{b} g_4$ se tedy dostáváme konfigurací do dvojice stavů (g_4, g_4), která náleží do bisimulační ekvivalence $\{(g_1, g_1), (g_2, g_2), (g_3, g_3), (g_4, g_4)\}$. Díky tomu, že jsou grafy bisimulačně ekvivalentní, bude takovýto tah vždy existovat alespoň jeden.

Dále uživateli zobrazíme tabulku s historií konfigurací hry (viz Obrázek 10), ve které se uživatel může, kliknutím na řádek tabulky, vrátit do předchozích konfigurací hry, změnit své rozhodnutí a pokračovat v hraní jinou cestou.

4.4.3 Výherní strategie útočníka

Opačným případem je výherní strategie útočníka v případě, že 2 LTS grafy zvolené uživatelem nejsou bisimulačně ekvivalentní. V této hře musí náš algoritmus vždy zvolit tah, kterým se vždy přiblíží k výhře útočníka, z čehož vyplývá, že budeme potřebovat možné kombinace konfigurace hry nějakým způsobem očíslovat tak, abychom věděli, za kolik tahů můžeme při každém rozhodnutí vyhrát a tuto hodnotu se snažit zmenšovat a tím dojít k výhře. Také nesmíme dovolit, aby se algoritmus zacyklil v konfiguracích hry a opakoval se a tím pádem nikdy nevyhrál.

Jedním z možných řešení je přidávat si k dvojicím stavů informace, které můžeme následně použít ve hře. Algoritmus, kterým zjišťujeme bisimilaritu, začíná s kombinací všech stavů a postupnou eliminací těchto párů stavů se snažíme dojít k množině bisimulační ekvivalence. V případě, že grafy nejsou bisimulačně ekvivalentní, tato množina se bude zmenšovat do doby, než zůstane prázdná. Vyřazování se vždy děje ve vyřazovacích kolech, kdy v našem případě jedna nebo více dvojic jsou vyřazeny. V našem algoritmu budeme počítat číslo vyřazovacího kola po-

Player	Alg
s: (s1)	u1: (u1)
s: (s1)-b->(s2)	u1: (u1)-b->(u2)
s: (s2)-a->(s)	u1: (u2)-a->(u)
s: (s)-a->(s1)	u1: (u)-a->(u1)
s: (s1)-b->(s2)	u1: (u1)-b->(u2)
s: (s2)-a->(s)	u1: (u2)-a->(u)
s: (s)-a->(s1)	u1: (u)-a->(u1)

Obrázek 10: Obrázek tabulky historie bisimulační hry

činaje číslem 0 a toto číslo připisovat k vyřazeným dvojicím. Například pokud dvojice (s_1, t_2) je vyloučena v prvním kole, uložíme si ji do množiny vyřazených dvojic společně s číslem kola, kdy byla vyřazena, to je číslo 0. Úprava našeho vyřazovacího algoritmu tedy spočívá v přidání tohoto počítadla. Vyřazovací algoritmus se skládá ze dvou hlavních cyklů. Vrchní cyklus porovnává velikost množiny dvojic stavů s předešlou množinou a opakuje, dokud se velikost těchto množin liší. V případě, že je velikost množin stejná, cyklus je ukončen, jelikož jsme buďto našli bisimulační ekvivalenci a nebo je množina prázdná. Tento cyklus tedy rozšíříme o počítadlo kol začínající na čísle 0 a inkrementující se po provedení těla cyklu. Druhý z hlavních cyklů je zanořen a prochází prvky naší množiny a kontroluje, jestli akce vedoucí z dvojice stavu vedou opět do dvojice stavů v množině. Pokud ano, ponecháme prvek v množině, v opačném případě prvek z množiny vyřadíme a společně s číslem vyřazovacího kola jej uložíme do množiny vyřazených stavů. Toto číslo nám pak značí kolik tahů obránce dovede zaručeně bránit. Pokud je číslo 0, znamená to, že z jednoho ze stavů vede akce taková, která neexistuje ve stavu druhém, to znamená, že útočník zahraje tuto hranu a obránce se již nemá jak bránit a prohrál, takže obránce má 0 tahů, ve kterých se dovede bránit. Pokud by hodnota byla 1, znamená to, že dvojice byla vyřazena protože vedla do dvojice, která již neexistovala v naší množině, to znamená, že vedla do některého ze stavů vyřazených v minulém kole, v tomto případě do stavů s vyřazovacím číslem 0. Obránci tedy zbývá jediný obranný tah a do stavů s číslem 0, ze kterých už nebude mít možnost táhnout.

Touto úpravou našeho vyřazovacího algoritmu jsme byli schopni označit všechny možné dvo-

jice stavů číslem označujícím, za kolik tahů může obránce prohrát a útočník vyhrát. Toto je naše hledané číslo, které můžeme využít v algoritmu útočníka tak, že se jej budeme snažit zmenšovat tak, abychom se jako útočník postupně dostali k výhře.

Samotný výběr grafu a útočného tahu z aktuální konfigurace hry probíhá následovně. Zjistíme jaké číslo má aktuální konfigurace přiřazeno. Zjistíme, kam můžeme na jednotlivých grafech z aktuální konfigurace hry táhnout a pro každý možný útočný tah zjistíme všechny možné tahy, které může obránce vykonat pro každý jednotlivý útok. Tímto dostaneme několik možných nových konfigurací hry pro tento útok a jeho možné obrany, pro každou konfiguraci nyní zjistíme číslo vyřazení. Nyní můžeme tatočísla porovnat s číslem aktuální konfigurace, a to tak, že od jednotlivých čísel možných nových konfigurací odečteme číslo aktuální konfigurace. Tímto nám pro každý možný útok a jeho obrany vznikne seznam čísel. Z každého tohoto seznamu čísel vybereme maximum a poté z těchto maxim vybereme minimum a útok vedoucí do seznamu, z kterého pochází toto minimum zvolíme. Tímto postupem zaručíme, že jakýkoliv tah obránce zvolí, vždy tento tah povede do menšího čísla vyřazení a tím se posunu k výhře. Z postupu vylučování možných konfigurací vyplývá, že toto číslo bude vždy -1 a vždy tam minimálně jedno takovéto číslo bude.

```
let setOfStates = [/* vsechny kombinace dvojic stavu grafu */]
let round = 0
let eliminatedStates = []

do {
    let setOfStatesSize = size of setOfStates
    for(statePair in setOfStates) {
        if(statePair má akci vedoucí mimo setOfStates) {
            přesun statePair z setOfStates do eliminatedStates a zaznamenej k ně
                mu aktuální hodnotu z proměnné round
        }
    }

    let newSetOfStatesSize = size of newSetOfStates
    setOfStates = newSetOfStates
    round++
} while(newSetOfStatesSize < setOfStatesSize)


let gameConfiguration = počátační stavy grafů

// volba tahu
while(je kam táhnout)
```

```
    let eliminationRound = zaznamenané vyřazovací kolo aktuální konfigurace
    let possibleMoves = možné konfigurace na které můžeme z aktuální
        konfigurace táhnout
    let moveResponsesGroups = možné konfigurace pro každý tah obránce z
        konfigurací v proměnné possibleMoves
    let groupMaximums = []
    for(groupOfMoves in moveResponsesGroups) {
        let moves = pro každou konfiguraci hry z groupOfMoves zjisti jeho vyř
            azovací kolo a dečti od něj eliminationRound
        přidej maximum z proměnné moves do proměnné groupMaximums
    }
    let moveToExecute = najdi konfiguraci s minimálním číslem vyřazovacího kola
         z proměnné groupMaximums
    gameConfiguration = konfigurace po provedení útočného tahu z possibleMoves,
         který vedl do konfigurace v proměnné moveToExecute
}
```

Výpis 6: Pseudokód výherní strategie útočníka

4.5 Vyhodnocování HML formulí

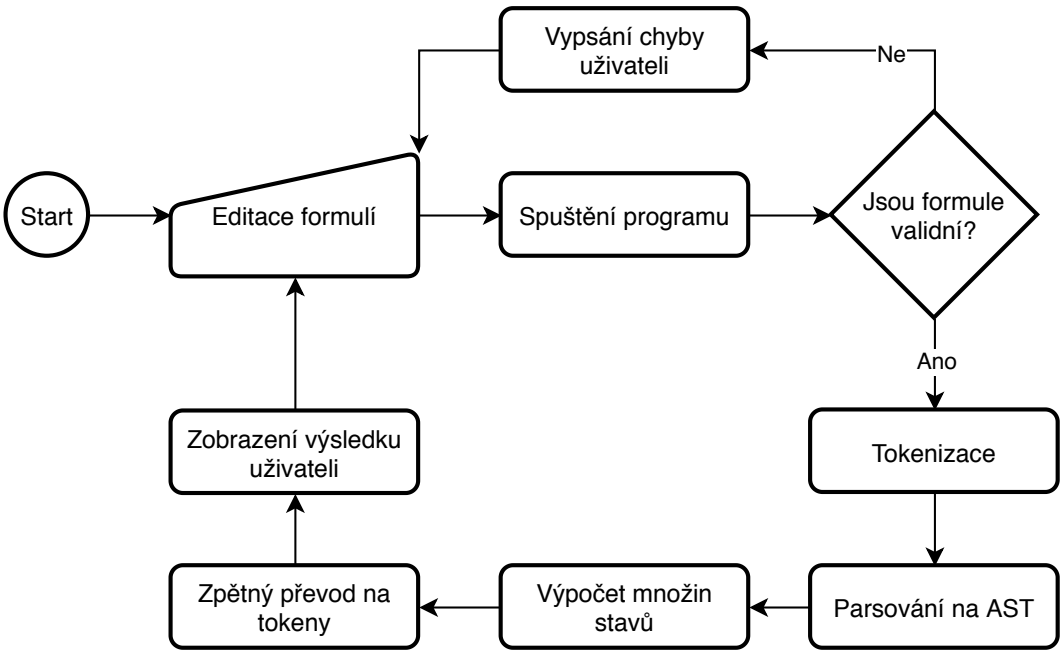
Pro vyhodnocení HML formulí s i bez rekurze je potřeba aplikovat několik kroků znázorněných v Obrázku 11. Jako první uživatel definuje jednu nebo více formulí, a to buďto s nebo bez rekurze, případně kombinaci obou. Nástroj poté zpracuje tyto formule a převede je na tokeny, tyto tokeny jsou dále převedeny na abstraktní syntaktický strom (AST), pro každý uzel v tomto stromu je poté vypočtena množina stavů, ve kterých je podvýraz splněn. Poté je AST převeden zpět na tokeny s příslušnými množinami stavů a tyto tokeny s množinami jsou vykresleny uživateli jako výsledek výpočtu.

4.5.1 Zadávání formulí s rekurzí uživatelem

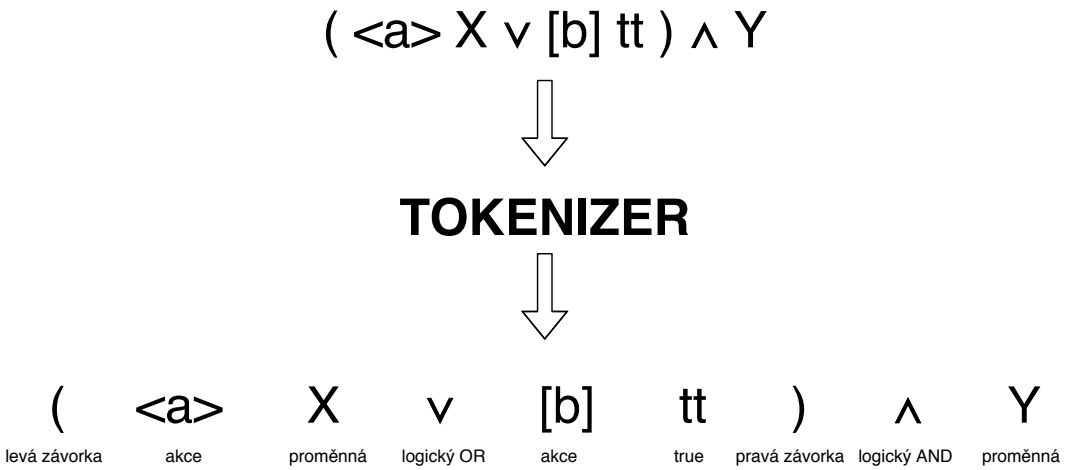
Pro uživatele je vytvořen textový editor umožnující zadávání HML formulí. Uživatel může formule zadávat do editoru dvojím způsobem.

1. Ručně zapsat HML formuli
2. Přidat již předem uloženou HML formuli

Jakmile všechny uživatelem zadané formule jsou validní HML formule, systém započne výpočet množin stavů, pro které tyto formule a jejich podformule platí.



Obrázek 11: Diagram postupu vyhodnocení HML formulí



Obrázek 12: Ukázka převodu výrazů na tokeny

4.5.2 Tokenizace formulí

Tokenizace[15] je rozdělování textu, v našem případě HML formule, do tokenů - nejmenších možných samostatných bloků. Proměnné, operátory, atp.

Jako první musíme definovat co považujeme za tokeny. Tato analýza nám dá hrubou představu, jak přesně bychom měli rozdělit HML formuli.

- Logický and operátor $\wedge$: Pro rozpoznání tohoto znaku nám postačí procházet text znak po znaku, avšak pro některé další víceznakové tokeny bude lepší použití regulárních výrazů.
- Logický or operátor: $\vee$.
- Závorky: ().
- Proměnné: Můžou se skládat jak z malých tak z velkých písmen a obsahovat číslice.
- Boolean: tt a ff.
- Akce: $\langle a \rangle$ a $[b]$.

Ideálně bychom chtěli do tokenizeru na vstupu poslat text a pravidla ve tvaru $\{key : RegExp, type\}$ a na výstupu dostat pole tokenů ve tvaru $\{type, value\}$.

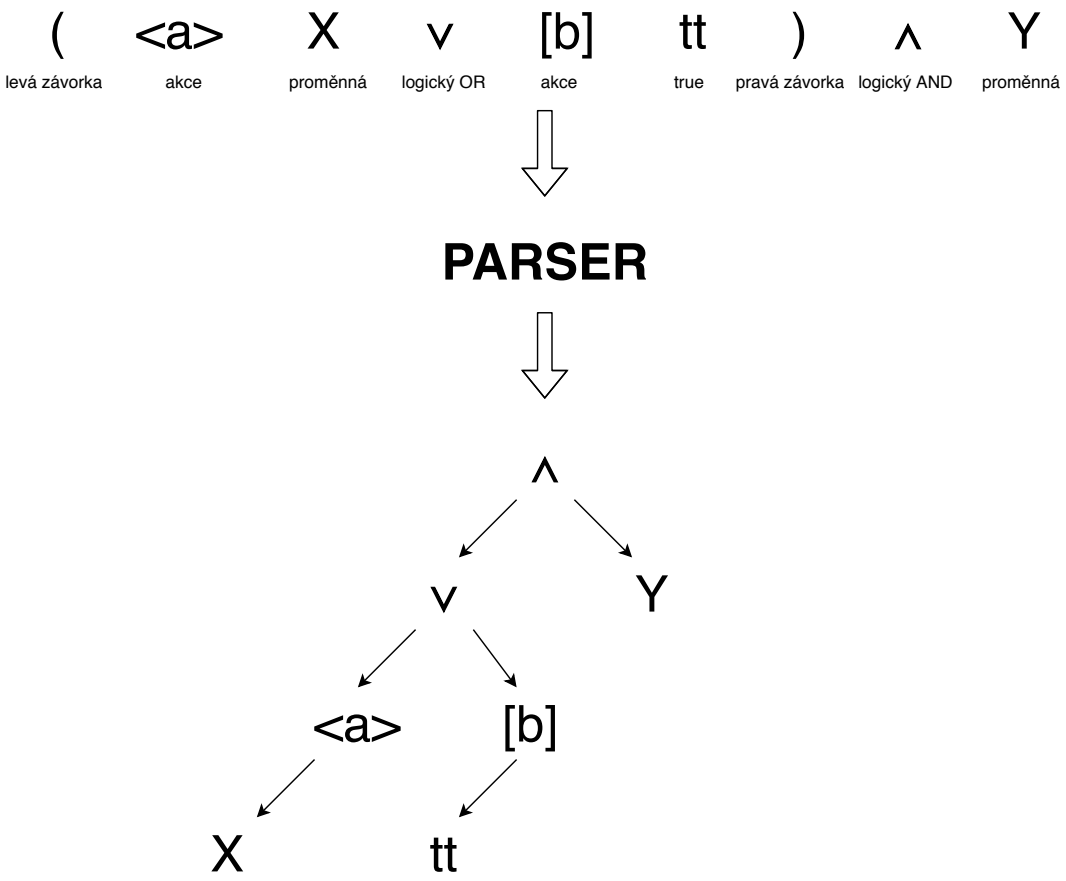
```
export const rules: Array<Rule> = [
  {
    key: /[  ]/g,
    type: TYPES.LAND,
  },
  {
    key: /[  ]/g,
    type: TYPES.LOR,
  },
  {
    key: /[(]/g,
    type: TYPES.LEFT_PARENTHESIS,
  },
  {
    key: /[)]/g,
    type: TYPES.RIGHT_PARENTHESIS,
  },
  {
    key: /<[a-zA-Z]*>/g,
    type: TYPES.SOME,
  },
```

```
  {
    key: /\[[a-zA-Z]*]/g,
    type: TYPES.EVERY,
  },
  {
    key: /tt/g,
    type: TYPES.TRUE,
  },
  {
    key: /ff/g,
    type: TYPES.FALSE,
  },
  {
    key: /[A-Z0-9]+/g,
    type: TYPES.VARIABLE,
  },
]
```

Výpis 7: Pravidla pro tokenizaci

Myšlenka implementace je rozdělovat text typu *String* postupně pomocí jednoho pravidla po druhém na pole obsahující typy *String* a *Token* tak, že pravidlu vyhovující místa v textu jsou převedeny na token a text je v tomto místě rozdělen na podřetězce, na kterých je pak testováno další pravidlo do doby, než vyčerpáme všechna pravidla. Problém ještě nastává u neviditelných znaků. Tyto znaky se dají buďto převést na speciální tokeny nebo zahodit. Jelikož neviditelné znaky nenesou žádnou zásadní informaci pro výpočet, rozhodl jsem se je zahodit. Dalším problémem je pořadí pravidel, jistá pravidla mohou částečně kolidovat s jinými pravidly. Například $\langle A \rangle$ může být označeno jako akce nebo jako proměnná A a další znaky, které nebudou žádným pravidlem zachyceny. Toto je nechtěné chování a dá se vyřešit několika způsoby, a to buďto pořadím pravidel (akce je rozpoznána jako první a převedena na token, tzn. pravidlo rozlišující proměnné už se na tento úsek nepodívá a nepřevede ho špatně) a nebo důmyslnějšími a složitějšími regulárními výrazy. V našem případě stačí zvolit vyhovující pořadí pravidel.

Mějme pravidla Výpis 7 a výraz $(X \vee \langle a \rangle tt) \wedge [b]ff$, průběh algoritmu je poté následující. Aplikujeme první pravidlo na logický and a dostaneme výsledek $[\text{"}(X \vee \langle a \rangle tt)\text{"}, \{type : \text{"}LAND\text{"}, value : \text{" } \wedge \text{ "}\}, \text{"}[b]ff\text{"}]$. Při dalším průchodu aplikujeme pravidlo na logický or, tímto dostaneme $[\text{"}(X\text{"}, \{type : \text{"}LOR\text{"}, value : \text{" } \vee \text{ "}\}\text{"}\langle a \rangle tt)\text{"}, \{type : \text{"}LAND\text{"}, value : \text{" } \wedge \text{ "}\}, \text{"}[b]ff\text{"}]$. Takto pokračujeme do konce seznamu pravidel, kde jako výsledek získáme následující seznam [
$\{type : \text{"}LEFT\_PARANTHESIS\text{"}, value : \text{"(}\text{"}\}$,
$\{type : \text{"}VARIABLE\text{"}, value : \text{"}X\text{"}\}$,
$\{type : \text{"}LOR\text{"}, value : \text{" } \vee \text{ "}\}$,



Obrázek 13: Ukázka převodu seznamu tokenů na AST

$\{type:\ "SOME",\ value:\ «a>"\},$
$\{type:\ "TRUE",\ value:\ "tt"\},$
$\{type:\ "RIGHT\_PARANTHESIS",\ value:\ ")"\},$
$\{type:\ "LAND",\ value:\ "\wedge"\},$
$\{type:\ "EVERY",\ value:\ "[b]"\},$
$\{type:\ "FALSE",\ value:\ "ff"\}$
].

4.5.3 Abstraktní syntaktický strom

V předchozí sekci jsme převedli HML formuli z textové podoby na pole tokenů, dalším krokem je použití parseru, který převede toto pole tokenů na abstraktní syntaktický strom[16]. Co se zapisování výrazů týče, existují 3 hlavní formy:

- infixová forma - ta na kterou jsme zvyklí $A \wedge B \vee C$
- postfixová forma - $AB \wedge C \vee$

- prefixová forma - $\vee C \wedge AB$

Tento údaj nám poskytne informaci o tom, jak výraz interpretovat správně (přednost a asociativita). Infix forma je velmi pohodlná pro čtení lidmi, avšak pro počítače je velmi nepřirozená. u postfix notace nebo abstraktního syntaktického stromu je pro počítač mnohem jednodušší s formulí pracovat. A to je místo, kde přichází na řadu parser, jeho úlohou je převést formuli v infix notaci na AST.

Jako první je potřeba rozšířit data pravidel z tokenizeru o přednosti a asociativitě jednotlivých pravidel. Takto dostaneme tokeny se strukturou $\{type, value, precedence, args\}$. Pro vytvoření AST z tokenů použijeme Shunting-yard algoritmus, jehož autorem je Dijkstra. Shunting-yard je metoda pro parsování matematických výrazů definovaných v infix notaci[17, 18]. Může produkovat buďto postfix notaci nebo AST. Algoritmus je založený na zásobníku, do kterého vkládáme operátory.

Příklad konverze:

1. Vstup: 1 * 2 + 3
2. Číslo 1 dáme na výstupní frontu
3. Operátor * dáme na zásobník
4. Číslo 2 dáme na výstupní frontu
5. Dokud je na zásobníku operátor s vyšší předností, přenášíme je na výstupní frontu, v našem případě operátor *
6. Operátor + dáme na zásobník
7. Číslo 3 dáme na výstupní frontu
8. Po přečtení výrazu, dáme operátory ze zásobníku do výstupní fronty
9. Výstup: 1 2 * 3 +

4.5.4 Výpočet množin stavů

Cílem výpočtu množin stavů je pro každý uzel v AST vyhodnotit všechny vyhovující stavy grafu tak, abychom mohli vyhodnotit celý výraz. Vyhodnocování máme dvojího druhu, a to s rekurzí a bez rekurze. Bez rekurze můžeme začít s vyhodnocováním od listů v AST, a to tak, že:

- Pokud token je typu true (tt), vyhovujícími stavy je množina všech možných stavů vyhodnocovaného grafu.
- Pokud token je typu false (ff), vyhovujícími stavy je prázdná množina.

- Pokud je token typu proměnná, vyhovujícími stavy je množina stavů vypočtena pro definici této proměnné.

Poté od vypočtených listů pokračujeme výpočty směrem ke kořenu stromu, kde se můžeme setkat s následujícími případy:

- Pokud token je typu $\langle a \rangle$, kde a náleží do Act a máme-li množinu stavů potomka S, vyhovujícími stavy je množina všech stavů S' obsahující stavy takové, do kterých vede nějaká hrana a ze stavů S.
- Pokud token je typu $[a]$, kde a náleží do Act a máme-li množinu stavů potomka S, vyhovujícími stavy je množina všech stavů S', která vychází ze stavů S, takových, pro které platí, že všechny hrany a ze stavu $s \in S$ vedou do S'.
- Pokud je token typu logického OR, vyhovujícími stavy je množina stavů vypočtena sjednocením množin potomků uzlu.
- Pokud je token typu logického AND, vyhovujícími stavy je množina stavů vypočtena průnikem množin potomků uzlu.

A takto vypočteme množiny pro celý AST v případě, že formule neobsahují rekurzi. V případě výpočtu s rekurzí je algoritmus potřeba pozměnit. Nejprve je potřeba definovat počáteční množiny pro všechny použité proměnné, a to následovně:

- Pokud definice proměnné neobsahuje rekurzi, vypočítej množinu podle algoritmu bez rekurze.
- Pokud je proměnná definována jako minimální, zvolíme počáteční množinu jako množinu všech stavů grafu.
- Pokud je proměnná definována jako maximální, zvolíme počáteční množinu jako prázdnou množinu.

Dále pokračujeme podobně jako v případě bez rekurze.

- Pokud token je typu true (tt), vyhovujícími stavy je množina všech možných stavů vyhodnocovaného grafu.
- Pokud token je typu false (ff), vyhovujícími stavy je prázdná množina.
- Pokud je token typu proměnná, vyhovujícími stavy je množina stavů vypočtena pro definici této proměnné v předchozím výpočtu.

Poté od vypočtených listů opět pokračujeme výpočty směrem ke kořenu stromu, kde se můžeme setkat s následujícími případy:

- Pokud token je typu $\langle a \rangle$, kde a náleží do Act a máme-li množinu stavů potomka S, vyhovujícími stavy je množina všech stavů S' obsahující stavy takové, do kterých vede nějaká hrana a ze stavů S.
- Pokud token je typu $[a]$, kde a náleží do Act a máme-li množinu stavů potomka S, vyhovujícími stavy je množina všech stavů S', která vychází ze stavů S, takových, pro které platí, že všechny hrany a ze stavu $s \in S$ vedou do S'.
- Pokud je token typu logického OR, vyhovujícími stavy je množina stavů vypočtena sjednocením množin potomků uzlu.
- Pokud je token typu logického AND, vyhovujícími stavy je množina stavů vypočtena průnikem množin potomků uzlu.

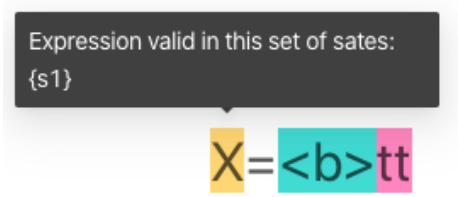
Jakmile vypočteme množiny pro jakoukoliv proměnnou, aktualizujeme její uložený výsledek tak, abychom mohli jej mohli použít v dalších výpočtech množin. Jakmile provedeme výpočty pro všechny proměnné, tak pokud všechny výpočty byly stejné s předchozí definicí výsledků, algoritmus končí a máme výsledné množiny. Pokud ovšem se výsledek některé proměnné lišil, je potřeba provést výpočty znova do té doby, než dojdeme do stavu, kdy se neliší a díky tomu, že tyto funkce jsou monotónní, nedojde k zacyklení.

4.5.5 Zpětný převod na tokeny

Nyní potřebujeme převést naplněný AST zpět na seznam tokenů rozšířený o množiny vypočtených stavů. Jelikož náš AST vznikl převodem ze seznamu tokenů a převodem vznikou opět stejné tokeny, je jedním z řešení si zapamatovat vazbu mezi tokeny a uzly v AST a poté pro každý token zpětně dohledat tento uzel v AST a překopírovat množinu stavů. Výsledná struktura tokenu bude rozšířena o stavy, pro které tento token platí $\{type,\ value,\ precedence,\ args,\ states\}$. Tento algoritmus není nijak efektivní, s asymptotickou časovou složitostí $O(n^2)$, kde n je počet uzlů v AST. Ovšem vzhledem k velikosti našich AST grafů, které budou v řadách jednotek až desítek uzlů, je tento algoritmus dostačující. Výhodou tohoto algoritmu je také identičnost výsledných tokenů s počátečními tokeny. Seznam tokenů může na obsahovat oproti AST závorky, informační hodnota těchto závorek je do AST přenesena ovšem závorky samotné přeneseny nejsou. Ve chvíli, kdy uživatel zadá závorky na místě takovém, že se nemění přednosti ve výrazu, v AST nerozpoznáme, zdali tam tyto závorky byly nebo ne, jelikož ale zpětný převod začínáme již s celým seznamem původních tokenů, tyto bezvýznamné závorky uživateli ve výrazu zůstanou.

4.5.6 Vykreslení uživateli

Vykreslení výsledku je provedeno interaktivně, uživatel může kurzorem přejíždět přes jednotlivé části formule a pozorovat stavy vyhovující této části formule v tooltip komponentě, tak jak je ukázáno na obrázku 14.



Obrázek 14: Ukázka vykreslení výsledku výpočtu formule

4.6 Persistence dat

Dalším z požadavků na nástroj je také možnost ukládání grafů. Tento požadavek je řešen pomocí 2 rozličných způsobů, a to pomocí exportování grafu do souboru a jeho zpětné načtení do nástroje, a uložení grafu v nástroji a jeho persistence mezi sezeními uživatele.

```
const myBlob = new Blob(jsonString, {type : 'application/json'});
URL.createObjectURL(myBlob)
```

Výpis 8: Serializace souboru

```
function FileLoader() {
  function handleLoadFromFile(event) {
    event.target.files[0].text()
      .then((jsonString) => {
        const graph = JSON.parse(jsonString)
        // ... nahrání grafu do~UI editoru
      })
      .catch(console.error)
  }

  return (
    <input type="file" onChange={handleLoadFromFile} />
  )
}
```

Výpis 9: Nahrání souboru

Export a import grafu do souboru je řešen pomocí serializace objektu grafu do formátu JavaScript Object Notation, zkráceně JSON, který obsahuje data modelu jednoho grafu. O samotné uložení a načtení souboru na a z disku uživatele se stará prohlížeč, který tuto funkčnost poskytuje skrze své API Browser Object Model (BOM), tohoto může být docíleno například tak jako v ukázce kódu 8 a 9.

O persistenci vytvořených grafů mezi sezeními uživatele se stará funkcionalita modulu LocalStorage nabízené v BOM prohlížeče. LocalStorage je jednoduchá databáze typu klíč-hodnota,

která nemá žádnou časovou dobu expirace. Data jsou pak uložena na disku zařízení, persistence je tedy závislá na zařízení, na kterém je nástroj používán a data nejsou tímto způsobem přenositelná mezi zařízeními.

4.7 Nasazení nástroje

Nástroj je uživatelům distribuován v podobě webové stránky přes HTTPS protokol. Pro tuto distribuci je nutné zdrojový kód nástroje transpilovat a vytvořit distribuovatelný balík a tento balík poté distribuovat pomocí HTTP serveru. Proces transpilace a vytvoření balíku je částečně automatizováno pomocí NPM skriptu *npm build* a nasazení je plně automatizováno pomocí NPM a GitLab Pages.

4.7.1 Transpilace a vytvoření balíku

Předpoklady:

- NodeJS
- NPM (součástí NodeJS)
- GIT

Postup:

1. Pomocí příkazu *git clone* stáhneme kód ze vzdáleného git repozitáře.
2. Pomocí příkazu *npm install*, spuštěného ve složce s projektem, nainstalujeme požadované softwarové závislosti.
3. Pomocí příkazu *npm build*, vytvoříme distribuovatelný balík souborů ve složce *build*.

4.7.2 Automatizace nasazení

K nasazení je použito nástrojů GitLab Continuous Integration and Continuous Deployment (GitLab CI/CD) a GitLab Pages. Obojí je nastaveno pomocí skriptu 10 vytvořeného v souboru *.gitlab − ci.yml*.

```
image: node:13.12.0 # Použitá verze NodeJS

cache:
  paths:
    - node_modules/ # Uložení závislostí pro urychlení instalace knihoven

before_script:
  - npm install # Instalace závislostí
```

```
test:
  stage: test
  script:
    - CI=true npm test # Spuštění testů před nasazením

pages:
  stage: deploy
  script:
    - CI=true npm run build # Vytvoření balíku pro nasazení
    - rm -rf public # Vymazání starých souborů
    - mv build public # Přesunutí balíku do hostované složky
  artifacts:
    paths:
      - public # GitLab Pages hostují stránky z "public" složky
  only:
    - master # Spuštění nasazovacího skriptu pouze na "master" GIT větvi
```

Výpis 10: Konfigurace GitLab Pipeline pro nasazení na GitLab Pages

5 Vstupy do programu

V této kapitole se budeme zajímat vytvořením testovacích vstupů pro aplikaci. Tyto vstupy jsou přibaleny k aplikaci a slouží k otestování scénářů v aplikaci bez nutnosti uživatele zadávat vstupy vlastní. Každý uživatel pak může tyto vstupy lokálně rozšířit o další importováním souboru vstupů nebo vytvořením vlastních vstupů v editorech poskytnutými nástrojem.

5.1 Předem připravené vstupy

Předem připravené vstupy musí pokrýt základní scénáře na odzkoušení nástroje. Tyto scénáře jsou:

- Jednoduché bisimulačně ekvivalentní grafy bez cyklu.
- Jednoduché bisimulačně neekvivalentní grafy bez cyklu.
- Jednoduché bisimulačně ekvivalentní grafy s cyklem.
- Jednoduché bisimulačně neekvivalentní grafy s cyklem.
- Složitější bisimulačně ekvivalentní grafy.
- Složitější bisimulačně neekvivalentní grafy.
- Několik HML výrazů bez rekurze vytvořených s ohledem na tyto grafy.
- Několik HML výrazů s rekurzí vytvořených s ohledem na tyto grafy.

Minimum tvoří 3 jednoduché grafy bez cyklů, 3 jednoduché grafy s cyklem, 3 složitější grafy, 2 HML výrazy bez rekurze a 2 HML výrazy s rekurzí. Tyto grafy společně s několika dalšími jsou vytvořeny a přibaleny k distribuci nástroje jako JSON soubor. Tento soubor je stažen při načtení nástroje a grafy jsou pak dostupné pro uživatele. Tyto grafy je možné změnit a novou verzi distribuovat. Uživatelé pak budou mít tyto nové grafy dostupné.

5.2 Importování souboru

K importování souborů do nástroje jsou určeny souboru formátu JSON kódován v jazykové sadě UTF-8. Formát JSON je čitelný člověkem, tudíž je očekáváno, že tyto tyto soubory mohou vzniknout buďto exportováním z nástroje, manuálním napsáním souboru uživatelem, generováním souboru externím programem, nebo kombinací těchto přístupů. Příklad souboru k importování můžeme vidět ve výpisu 11. Soubor s grafem obsahuje 4 základní položky, a to:

- nodes - Pole objektů obsahující informaci o umístění jednotlivých uzlů grafu v kreslícím plátně.

- edges - Pole objektů obsahující informaci o propojení dvou uzlů a souřadnice pro kvadratickou Bézierovu křivku.
- start - Počáteční uzel grafu.
- name - Jméno grafu.

```
{
  "nodes": [
    { "label": "s", "x": 226, "y": 66 },
    { "label": "s1", "x": 217, "y": 204 },
    { "label": "s2", "x": 207, "y": 332 }
  ],
  "edges": [
    {
      "source": "s", "target": "s1", "label": "a", "quadraticCurve": {
        "x": 221,
        "y": 125
      }
    },
    {
      "source": "s1", "target": "s2", "label": "b", "quadraticCurve": {
        "x": 213,
        "y": 258.5
      }
    },
    {
      "source": "s2", "target": "s2", "label": "b", "quadraticCurve": {
        "x": 250,
        "y": 293
      }
    },
    {
      "source": "s2", "target": "s", "label": "a", "quadraticCurve": {
        "x": 300,
        "y": 202
      }
    }
  ],
  "start": "s",
```

```
  "name": "S",
}
```

Výpis 11: Příklad souboru k importu

5.3 Vytvoření vstupů v nástroji

Pro zadávání vstupů manuálně v nástroji slouží grafický editor grafů popsaný v kapitole 4.3 Zadávání LTS grafem. Tento editor může posloužit i pro editování již v minulosti vytvořených grafů včetně předem připravených grafů. Pro tuto možnost je potřeba kliknout na ikonu editace grafu v přehledu grafů, uživatel je pak přesměrován na editor s načteným grafem k editaci.

Editor grafu nabízí akci pro tvorbu uzlu grafu, tvorbu hrany mezi dvěma uzly nebo stejným uzlem. Tyto akce jsou spuštěny přes Alt+Klik myší (Command + Klik myší na MacOS), a to v případě tvorby uzlu, musí být klik proveden do prázdného prostoru a v případě tvorby hrany musí být klik proveden na uzel grafu. Další z možných akcí je přejmenování uzlů grafu a akcí na hranách, obojí je prováděno klikem na jméno uzlu či jméno akce. Akce je také pro přehlednost možné zadávat hromadně na jedné hraně, stačí akce oddělit čárkou, například a, b, c. Pro posunutí uzlu grafu je potřeba na uzel kliknout myší a táhnout na požadovanou pozici.

6 Závěr

Úspěšně se podařilo zpracovat a splnit všechny úkoly a cíle této diplomové práce dle zadání a vytvořit tak nástroj pro výukový server předmětů teoretické informatiky.

Prvním úkolem bylo nastudovat problematiku konečných ohodnocených přechodových systémů, bisimulační ekvivalence těchto systémů a Hennessy-Milner logiku s rekurzí. Zde jsem pochopil pojmy týkající se této problematiky a různá teoretická řešení těchto problémů.

Dále bylo potřeba se zamyslet nad samotnou implementací zadaného nástroje, specifikoval jsem proto funkční i nefunkční požadavky tohoto nástroje, podle kterých jsem se při dalším návrhu nástroje a samotné implementaci řídil. Zvolil jsem čistě klientskou aplikaci pro svou nenáročnost na centrální server, a to jak z výkonnostního hlediska, kdy všechny výpočty jsou prováděny na zařízení koncového uživatele, tak z technologického hlediska, kdy jediná podmínka pro distribuci aplikace je HTTP server. Dále jsem pokračoval samotnou implementací algoritmů pro výpočet bisimulační ekvivalence, bisimulačních her a výpočty HML formulí.

Dalším z cílů bylo vytvořit způsob pro zadávání nových konečných ohodnocených přechodových systémů do nástroje, vytváření a editace HML formulí a způsob interakce uživatele s nástrojem při hraní her bisimulační ekvivalence, a to jak pro bisimulačně ekvivalentní, tak bisimulačně neekvivalentní grafy. Dále bylo potřeba vytvořit systém pro ukládání přechodových systémů do souboru a zpětné načítání těchto souborů, obdobně ukládání a načítání do perzistentní paměti prohlížeče.

Posledním z cílů bylo vytvoření testovacích dat pro nástroj tak, aby si uživatel mohl odzkoušet celý nástroj i bez zadávání vlastních vstupů. Tato data jsou pak přibalena k distribuci nástroje a dostupné od prvního spuštění nástroje. Tato data je možno rozšířit a editovat JSON souboru a poté distribuovat.

Nástroj je možno dalším vývojem rozšířit nové vlastnosti, těmi může být například, hraní her slabé bisimilarity, možnosti zadávání systémů pomocí alternativních způsobů jako jsou jazyky procesní algebry, použití efektivnějších algoritmů pro výpočet bisimilarity, načítání připravených LTS grafů z jiných online zdrojů nebo možnost zadávání úkolů v nástroji.

Literatura

1. ACETO, Luca. *Reactive systems: modelling, specification and verification.* Cambridge University Press, 2007.
2. BURKART, Olaf; CAUCAL, Didier; MOLLER, Faron; STEFFEN, Bernhard. CHAPTER 9 - Verification on Infinite Structures. In: BERGSTRA, J.A.; PONSE, A.; SMOLKA, S.A. (ed.). *Handbook of Process Algebra.* Amsterdam: Elsevier Science, 2001, s. 545–623. ISBN 978-0-444-82830-9. Dostupné z DOI: `https://doi.org/10.1016/B978-044482830-9/50027-8`.
3. GLABBEEK, Robert Jan van. The Linear Time - Branching Time Spectrum I. *The Linear Time - Branching Time Spectrum I.* Dostupné z DOI: `10.1.1.121.9596`.
4. GRANAS, Andrzej; DUGUNDJI, James. *Fixed point theory.* Springer, 2003.
5. HENNESSY Matthew; Milner, Robin. *On observing nondeterminism and concurrency. Automata, Languages and Programming. Lecture Notes in Computer Science.* Springer, 1980.
6. *Standard ECMA-262.* Dostupné také z: `https://www.ecma-international.org/publications/standards/Ecma-262.htm`. [cit. 3.4.2020].
7. *JavaScript.* Dostupné také z: `https://developer.mozilla.org/en-US/docs/Web/JavaScript`. [cit. 3.4.2020].
8. *Typed JavaScript at Any Scale.* Dostupné také z: `https://www.typescriptlang.org/`. [cit. 3.4.2020].
9. *React – A JavaScript library for building user interfaces.* Dostupné také z: `https://reactjs.org/`. [cit. 3.4.2020].
10. *Babel - The compiler for next generation JavaScript.* Dostupné také z: `https://babeljs.io/`. [cit. 3.4.2020].
11. *Webpack documentation.* Dostupné také z: `https://webpack.js.org/concepts/`. [cit. 3.4.2020].
12. *What is npm?* Dostupné také z: `https://www.w3schools.com/whatis/whatis_npm.asp`. [cit. 3.4.2020].
13. KANELLAKIS, P. C.; SMOLKA, Scott Allen. *CCS expressions, finite state processes, and three problems of equivalence.* Brown University, Dept. of Computer Science, 1983.
14. PAIGE, Robert; TARJAN, Robert E. *Three partition refinement algorithms.* Princeton University, Dept. of Computer Science, 1986.
15. ESIMOV, Miras. *Math AST: Tokenizer in JavaScript.* Dostupné také z: `https://www.esimovmiras.cc/articles/02-build-math-ast-tokenizer/`. [cit. 3.4.2020].

16. ESIMOV, Miras. *Math AST: Parser in JavaScript*. Dostupné také z: https://www.esimovmiras.cc/articles/03-build-math-ast-parser/. [cit. 3.4.2020].

17. *Shunting Yard Algorithm*. Dostupné také z: https://brilliant.org/wiki/shunting-yard-algorithm/. [cit. 3.4.2020].

18. *Parsing/Shunting-yard algorithm*. Dostupné také z: https://rosettacode.org/wiki/Parsing/Shunting-yard_algorithm. [cit. 3.4.2020].

A Případy užití

A.1 Zadání LTS grafem

Hlavní aktéři: Uživatel systému, Systém.
Předpoklady: Žádné předpoklady.
Minimální výsledek: Nedojde k porušení dat v systému.
Úspěšný výsledek: V systému je vytvořen nový graf.

Hlavní úspěšný scénář:

1. Uživatel započne vytváření grafu.
2. Uživatel vytvoří graf v grafickém editoru.
3. Uživatel unikátně pojmenuje graf.
4. Uživatel uloží graf.
5. Systém přesměruje uživatele na seznam grafů.

Rozšíření:

2a. Uživatel nevytvoří graf.

2a1. Systém nedovolí uživateli graf uložit.

2b. Uživatel importuje graf ze souboru.

2b1. Systém načte a zobrazí graf.

3a. Uživatel nepojmenuje graf.

3a1. Systém vygeneruje unikátní jméno grafu pro uložení.

3b. Uživatel pojmenuje graf duplicitním jménem.

3b1. Systém upraví jméno tak, aby bylo unikátní.

3c. Uživatel uloží graf do souboru.

3c1. Systém poskytne uživateli soubor ke stažení obsahující data grafu.

A.2 Editace existujícího LTS

Hlavní aktéři: Uživatel systému, Systém.
Předpoklady: V systému existuje alespoň 1 graf.
Minimální výsledek: Nedojde k porušení dat v systému.
Úspěšný výsledek: Graf v systému je změněn.
Hlavní úspěšný scénář:

1. Uživatel započne editaci LTS grafu z přehledu grafů.
2. Uživatel upraví graf v grafickém editoru.
3. Uživatel uloží graf.
4. Systém přesměruje uživatele na seznam grafů.

Rozšíření:

2a. Uživatel neupraví graf.

2a1. Pokračuje se krokem 3.

2b. Uživatel importuje graf ze souboru.

2b1. Systém načte a zobrazí graf.

3a. Uživatel smazal jméno grafu.

3a1. Systém vygeneruje unikátní jméno grafu pro uložení.

3b. Uživatel pojmenuje graf duplicitním jménem.

3b1. Systém upraví jméno tak, aby bylo unikátní.

3c. Uživatel uloží graf do souboru.

3c1. Systém poskytne uživateli soubor ke stažení obsahující data grafu.

A.3 Zobrazení bisimilarity LTS

Hlavní aktéři: Uživatel systému, Systém.
Předpoklady: V systému existuje alespoň jeden graf.
Minimální výsledek: Nedojde k porušení dat v systému.
Úspěšný výsledek: Hráč zahraje bisimulační hru.

Hlavní úspěšný scénář:

1. Uživatel započne ověřování bisimilarity.
2. Uživatel vybere 2 grafy.
3. Systém vyhodnotí bisimilaritu.

Rozšíření:

3a. Grafy jsou bisimulačně ekvivalentní.

3a1. Systém zobrazí množinu bisimulačních stavů grafů.

3a2. Uživatel vybere graf ve, kterém chce táhnout a zahraje útočný tah.

3a3. Systém zahraje obraný tah.

3a4a. Uživateli nezbývá žádný tah.

3a4a1. Uživateli je nabídnuto vrátit se k předchozím tahům.

3a4a2. Uživatel zvolá z předchozích tahů.

3a4a3. Systém navrátí stav to zvoleného stavu.

3a4a4. Uživatel pokračuje krokem 3a2.

3a4b. Uživatel pokračuje krokem 3a2.

3b. Grafy nejsou bisimulačně ekvivalentní.

3b1a. Systém zahraje útočný tah.

3b1b. Systému nezbývá žádný tah.

3b1b1. Systém ukončí hru.

3b2a. Uživatel zahraje obranný tah.

3b2b. Uživateli nezbývá žádný tah.

3b2b1. Uživateli je nabídnuto vrátit se k předchozím tahům.

A.4 Vypočtení HML výrazů

Hlavní aktéři: Uživatel systému, Systém.
Předpoklady: V systému existuje alespoň jeden graf.
Minimální výsledek: Nedojde k porušení dat v systému.
Úspěšný výsledek: Je zobrazen zadaný výraz a množiny stavů.

Hlavní úspěšný scénář:

1. Uživatel vybere graf pro výpočet HML výrazů.
2. Uživatel zadá validní HML výraz.

3. Systém zobrazí množiny stavů, pro které platí jednotlivé podformule a celá formule.

Rozšíření:

2a. Uživatel zadá nevalidní HML výraz

2a1. Systém zobrazí hlášku o invaliditě HML výrazu.

2a2. Uživatel opraví výraz na validní HML výraz.

2a3. Scénář pokračuje krokem 3.

B Seznam elektronických příloh

- Program
  - zdrojovy_kod_vas0122.zip - Archiv obsajující zdrojový kód aplikace.
  - distribucni_balik_vas0122.zip - Archiv obsajující zkompilovan kód aplikace připravený pro distribuci.
  - manual_vas0122.md - Soubor obsahující textový popis pro kompilaci aplikace.
- Text
  - diplomova_prace_vas0122.pdf - Soubor obsahující text diplomové práce.